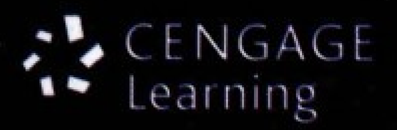
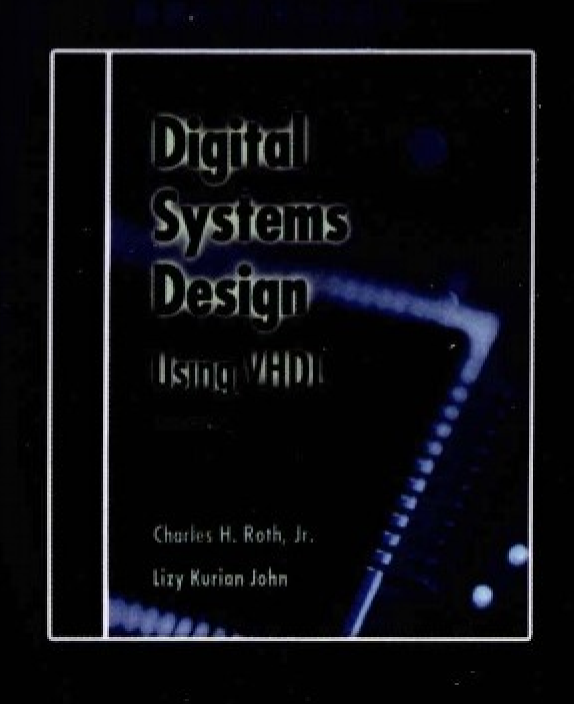
国外电子与通信教材系列



数字系统设计与VHDL

（第二版）

Digital Systems Design Using VHDL, Second Edition



[美] Charles H. Roth, Jr.
Lizy Kurian John 著
金明录 刘倩 译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

数字系统设计与VHDL（第二版）

Digital Systems Design Using VHDL, Second Edition

本书是作者多年来在得克萨斯大学奥斯汀分校讲授数字系统设计课程的经验积累，介绍了数字系统设计的基本原理及VHDL语言在数字系统设计中的运用，并把VHDL语言和数字系统设计融为一体。

本书特点

- 重点讲解基于VHDL的数字系统设计的过程，而不是纯粹讲解VHDL语言
- 讲解了许多设计例子，从设计简单的加法器到复杂的微处理器，便于教师根据具体教学要求进行选择
- 书中所有VHDL程序代码都使用IEEE的标准库文件，并通过了ModelSim仿真测试
- 本书配有教辅材料与习题解答

作者简介

Charles H. Roth, Jr.：分别在明尼苏达大学、麻省理工学院和斯坦福大学获得电子工程专业本科、硕士和博士学位，1961年就职于得克萨斯大学奥斯汀分校，目前是电气与计算机工程系的教授。Roth博士曾开发了逻辑设计课程的自学平台，因其出色的工程教育模式获General Dynamics Award奖。他的授课和研究领域涵盖了数字系统理论和设计、微计算机系统和VHDL应用，出版了4本著作。

Lizy Kurian John：在宾夕法尼亚州立大学获得计算机工程专业博士学位，1996年就职于得克萨斯大学奥斯汀分校。她的教学和科研主要涉及数字设计、计算机体系结构、微处理器和存储系统设计与性能评价。



CENGAGE Learning
www.cengageasia.com



责任编辑：谭海平
段丹辉
责任美编：喻　晓



ISBN 978-7-121-06728-0
9 787121 067280 >
定价：55.00 元

国外电子与通信教材系列

数字系统设计与VHDL

（第二版）

Digital Systems Design Using VHDL, Second Edition

［美］Charles H. Roth, Jr.　Lizy Kurian John　著

金明录　刘　倩　译

電子工業出版社

Publishing House of Electronics Industry

北京・BEIJING

内 容 简 介

本书是为本科生和研究生撰写的数字系统设计高级课程教材，它把工业标准硬件描述语言 VHDL 和数字系统设计融为一体。作者首先复习了数字系统设计的基本原理，然后从 VHDL 语言的基础知识开始，覆盖了许多基于 VHDL 语言的数字系统设计高级专题。学生了解基本原理之后，学习数字系统设计的最好方法是通过实际例子。因此本书中包含了丰富的设计实例，从简单的二进制加法器到复杂的微处理机设计，书中都进行了详细的介绍。本书的最大特点不是把 VHDL 语言作为单纯的程序语言来讲解，而是把重点放在 VHDL 语言在数字系统设计中的实际应用上。

本书可作为高等院校电子、电气和计算机专业本科生、硕士生的教材，也可作为相关工程技术人员的参考书。

Digital Systems Design Using VHDL，Second Edition

Charles H. Roth, Jr. , Lizy Kurian John

ISBN 978-0-534-38462-3

版权贸易合同登记号 图字：01-2008-1830

图书在版编目（CIP）数据

数字系统设计与 VHDL：第二版 /（美）罗斯（Roth, C. H.）著；金明录，刘倩译. —北京：电子工业出版社，2008.8
（国外电子与通信教材系列）

书名原文：Digital Systems Design Using VHDL，2E

ISBN 978-7-121-06728-0

I. 数… II. ①罗… ②金… ③刘… III. ①数字系统－系统设计－教材 ②硬件描述语言，VHDL－教材

IV. TP271 TP312

中国版本图书馆 CIP 数据核字（2008）第 072162 号

责任编辑：谭海平　段丹辉
印　　刷：北京东光印刷厂
装　　订：三河市鹏成印业有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：29.75　字数：818.7 千字
印　　次：2008 年 8 月第 1 次印刷
定　　价：55.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

序

2001年7月间，电子工业出版社的领导同志邀请各高校十几位通信领域方面的老师，商量引进国外教材问题。与会同志对出版社提出的计划十分赞同，大家认为，这对我国通信事业、特别是对高等院校通信学科的教学工作会很有好处。

教材建设是高校教学建设的主要内容之一。编写、出版一本好的教材，意味着开设了一门好的课程，甚至可能预示着一个崭新学科的诞生。20世纪40年代MIT林肯实验室出版的一套28本雷达丛书，对近代电子学科、特别是对雷达技术的推动作用，就是一个很好的例子。

我国领导部门对教材建设一直非常重视。20世纪80年代，在原教委教材编审委员会的领导下，汇集了高等院校几百位富有教学经验的专家，编写、出版了一大批教材；很多院校还根据学校的特点和需要，陆续编写了大量的讲义和参考书。这些教材对高校的教学工作发挥了极好的作用。近年来，随着教学改革不断深入和科学技术的飞速进步，有的教材内容已比较陈旧、落后，难以适应教学的要求，特别是在电子学和通信技术发展神速、可以讲是日新月异的今天，如何适应这种情况，更是一个必须认真考虑的问题。解决这个问题，除了依靠高校的老师和专家撰写新的符合要求的教科书外，引进和出版一些国外优秀电子与通信教材，尤其是有选择地引进一批英文原版教材，是会有好处的。

一年多来，电子工业出版社为此做了很多工作。他们成立了一个“国外电子与通信教材系列”项目组，选派了富有经验的业务骨干负责有关工作，收集了230余种通信教材和参考书的详细资料，调来了100余种原版教材样书，依靠由20余位专家组成的出版委员会，从中精选了40多种，内容丰富，覆盖了电路理论与应用、信号与系统、数字信号处理、微电子、通信系统、电磁场与微波等方面，既可作为通信专业本科生和研究生的教学用书，也可作为有关专业人员的参考材料。此外，这批教材，有的翻译为中文，还有部分教材直接影印出版，以供教师用英语直接授课。希望这些教材的引进和出版对高校通信教学和教材改革能起一定作用。

在这里，我还要感谢参加工作的各位教授、专家、老师与参加翻译、编辑和出版的同志们。各位专家认真负责、严谨细致、不辞辛劳、不怕琐碎和精益求精的态度，充分体现了中国教育工作者和出版工作者的良好美德。

随着我国经济建设的发展和科学技术的不断进步，对高校教学工作会不断提出新的要求和希望。我想，无论如何，要做好引进国外教材的工作，一定要联系我国的实际。教材和学术专著不同，既要注意科学性、学术性，也要重视可读性，要深入浅出，便于读者自学；引进的教材要适应高校教学改革的需要，针对目前一些教材内容较为陈旧的问题，有目的地引进一些先进的和正在发展中的交叉学科的参考书；要与国内出版的教材相配套，安排好出版英文原版教材和翻译教材的比例。我们努力使这套教材能尽量满足上述要求，希望它们能放在学生们的课桌上，发挥一定的作用。

最后，预祝“国外电子与通信教材系列”项目取得成功，为我国电子与通信教学和通信产业的发展培土施肥。也恳切希望读者能对这些书籍的不足之处、特别是翻译中存在的问题，提出意见和建议，以便再版时更正。

吴佑寿

中国工程院院士、清华大学教授

“国外电子与通信教材系列”出版委员会主任

出 版 说 明

进入21世纪以来，我国信息产业在生产和科研方面都大大加快了发展速度，并已成为国民经济发展的支柱产业之一。但是，与世界上其他信息产业发达的国家相比，我国在技术开发、教育培训等方面都还存在着较大的差距。特别是在加入WTO后的今天，我国信息产业面临着国外竞争对手的严峻挑战。

作为我国信息产业的专业科技出版社，我们始终关注着全球电子信息技术的发展方向，始终把引进国外优秀电子与通信信息技术教材和专业书籍放在我们工作的重要位置上。在2000年至2001年间，我社先后从世界著名出版公司引进出版了40余种教材，形成了一套“国外计算机科学教材系列”，在全国高校以及科研部门中受到了欢迎和好评，得到了计算机领域的广大教师与科研工作者的充分肯定。

引进和出版一些国外优秀电子与通信教材，尤其是有选择地引进一批英文原版教材，将有助于我国信息产业培养具有国际竞争能力的技术人才，也将有助于我国国内在电子与通信教学工作中掌握和跟踪国际发展水平。根据国内信息产业的现状、教育部《关于“十五”期间普通高等教育教材建设与改革的意见》的指示精神以及高等院校老师们反映的各种意见，我们决定引进“国外电子与通信教材系列”，并随后开展了大量准备工作。此次引进的国外电子与通信教材均来自国际著名出版商，其中影印教材约占一半。教材内容涉及的学科方向包括电路理论与应用、信号与系统、数字信号处理、微电子、通信系统、电磁场与微波等，其中既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。我们还将与国外出版商一起，陆续推出一些教材的教学支持资料，为授课教师提供帮助。

此外，“国外电子与通信教材系列”的引进和出版工作得到了教育部高等教育司的大力支持和帮助，其中的部分引进教材已通过“教育部高等学校电子信息科学与工程类专业教学指导委员会”的审核，并得到教育部高等教育司的批准，纳入了“教育部高等教育司推荐——国外优秀信息科学与技术系列教学用书”。

为做好该系列教材的翻译工作，我们聘请了清华大学、北京大学、北京邮电大学、南京邮电大学、东南大学、西安交通大学、天津大学、西安电子科技大学、电子科技大学、中山大学、哈尔滨工业大学、西南交通大学等著名高校的教授和骨干教师参与教材的翻译和审校工作。许多教授在国内电子与通信专业领域享有较高的声望，具有丰富的教学经验，他们的渊博学识从根本上保证了教材的翻译质量和专业学术方面的严格与准确。我们在此对他们的辛勤工作与贡献表示衷心的感谢。此外，对于编辑的选择，我们达到了专业对口；对于从英文原书中发现的错误，我们通过与作者联络、从网上下载勘误表等方式，逐一进行了修订；同时，我们对审校、排版、印制质量进行了严格把关。

今后，我们将进一步加强同各高校教师的密切关系，努力引进更多的国外优秀教材和教学参考书，为我国电子与通信教材达到世界先进水平而努力。由于我们对国内外电子与通信教育的发展仍存在一些认识上的不足，在选题、翻译、出版等方面的工作中还有许多需要改进的地方，恳请广大师生和读者提出批评及建议。

电子工业出版社

教材出版委员会

译 者 序

数字系统设计和 VHDL 语言方面的书籍目前很多，但是 Charles H. Roth, Jr.教授的 *Digital Systems Design Using VHDL, Second Editon* 以特有的魅力，在本科生和研究生的数字系统设计高级课程中作为教材广泛被采用，它把工业标准硬件描述语言 VHDL 和数字系统设计融为一体。

译者于 1999 年在国外第一次接触到 VHDL 的时候，该书为译者打下了非常好的基础，在以后的数字系统设计中受益匪浅。回国后，译者一直从事于数字系统设计的教学和科研，参加过许多本科生和研究生的毕业答辩，非常吃惊地发现许多同学还没有掌握 VHDL 语言的精髓，很多设计都是照葫芦画瓢，追踪其因主要是把 VHDL 语言仅仅作为一种程序语言来学习，对 VHDL 语言的硬件描述的本质还不清楚，经常是为了编程而编程。

本书精细组织安排了教材内容和习题，有助于克服上述问题。本书具有以下几个特点：

1. 介绍基于 VHDL 的高级数字系统设计的过程，把数字设计概念与 VHDL 语言有机的结合起来，重点在于数字系统的设计，而不是为了纯粹的 VHDL 语言的介绍。
2. 介绍基于 VHDL 的数字模型、模拟仿真和数字系统的综合方法。教材中较早地介绍逻辑电路综合过程，同时强调了可综合 VHDL 代码的编写。
3. 介绍了许多设计例子，从简单的加法器到复杂的微程序和 MIPS 处理器的设计。这样教师可以根据具体教学要求选择最适合的例子用于教学。
4. 精细组织安排了一些例子和不同难度的练习题，有助于学生巩固基本概念。
5. 教材内容组织注意一般性，同时把实际具体产品作为参考例子，突出基于可编程器件来实现的基本原理。
6. 教材中的所有 VHDL 程序代码都通过了提供了 ModelSim 软件的仿真测试。ModelSim 软件的学生版本对学生是免费的可以通过网站连接下载。
7. 教材自始至终基于 IEEE 标准和库文件进行设计。
8. 包含教材中所有章节图片的讲稿可以通过网站下载。

本书译稿由大连理工大学电信学院的金明录教授和刘倩负责完成。刘倩、闫肃、林晓峰、刘来增、黄昊、郭猛、李德峰、曲宏伟和董香花等同学和徐钢讲师曾参与了本书第一版的部分章节的翻译工作，在此表示感谢。另外，感谢电子工业出版社的马岚、谭海平和段丹辉等编辑在组织出版和编辑工作中所给予的支持。

限于译者水平，书中难免有错误与不妥之处，敬请广大读者批评指正。

前　言

本书可以作为大学高年级数字系统设计课程的教科书。书中介绍了数字系统设计的基本原理及其在设计中硬件描述语言（VHDL）的运用。介绍基本原理之后，设计方法最好通过例子讲授。为此书中包含了丰富的设计实例，从简单的二进制加法器到复杂的微处理器的设计。

使用本教科书之前，学生最好完成逻辑设计的基础课程，包括组合电路和时序电路设计。虽然不要求具有 VHDL 语言的基础知识，但是学生最好具有使用现代高级语言（如 C 语言）的编程经验。如果学生已经学过汇编语言编程和计算机体系结构方面的课程，那么对于加深理解本书中的内容（特别是第 9 章）也是很有帮助的。

因为一般学生在大学期间的两年以前就已完成了逻辑设计的基础课程，所以大多数的学生需要复习基础知识。因此第 1 章回顾了逻辑设计的基本原理。由于大多数的学生可以自行完成本章的复习，所以教师可以不用在本章花费太多的时间。然而，对时序电路的时序和同步设计原理的较好掌握，是数字系统设计过程的关键所在。

第 2 章先回顾了现代设计流程，同时归纳了数字设计实现的各种技术，接着介绍了 VHDL 语言的基础知识。在下面的章节中将使用这些 VHDL 硬件描述语言进行设计，VHDL 语言的其他功能根据需要逐渐引入，而更多的高级功能将在第 8 章中介绍。我们一开始就把 VHDL 的编程和相应硬件联系起来。现在有些教材把 VHDL 只作为编程语言讲授，把主要篇幅放在介绍 VHDL 语言的语法上。与此正相反，我们则把重点放在介绍如何在数字系统设计过程中使用 VHDL 语言。这门语言其实很复杂，所以我们不涵盖其所有内容。我们重点介绍了数字系统设计所需的主要功能，而忽略了其他不常使用的功能。本章中还介绍了标准 IEEE VHDL 库文件的使用，在书中自始至终我们只使用了 IEEE 标准库文件。

VHDL 语言在讲授自上而下（top-down）的设计中非常有用。我们可以在顶层设计一个系统并可以用 VHDL 描述算法，并且在进行具体的逻辑设计之前对所设计的系统可以仿真和调试。但是任何一个设计最终都以实际硬件实现，并且在其通过测试后，才算告终。为此，我们建议本课程包含一些硬件实现的设计实验。在第 3 章，我们介绍了简单的可编程逻辑器件（PLD），以便在课程中根据需要较早使用实际硬件。在第 3 章先回顾了可编程逻辑器件，接着介绍了简单的可编程逻辑器件，随后介绍了复杂可编程逻辑器件（CPLD）和现场可编程门阵列（FPGA）。现在市面上有很多相关产品，让学生们了解一些商用器件是有益的，然而更重要的是要让学生了解构造这些可编程器件的基本原理。因此，在本章中我们主要介绍这些器件的一般性原理，而具体的器件只作例子给出。这一章的内容也为第 6 章中 FPGA 较为深入的讨论打下基础。

第 4 章介绍了很多设计实例，包括算术和非算术实例。本章包含了从简单的 BCD-七段数码管显示译码器到更复杂的像记分板电路、键盘扫描器和二进制除法器等例子。本章中还给出了用于算术运算的一般技术，如先行进位加法、二进制乘法和除法。用状态机控制数字系统的操作顺序是本章中介绍的一个重要概念。同时，本章中还给出了各种可综合的 VHDL 代码。本章中介绍了大量的实例，教师可以根据自己的喜好选择适当的实例讲授给学生。

第 5 章介绍了与状态图一样有用的流程机图(SM 图)。我们介绍了如何基于 SM 图进行 VHDL 编程设计，以及如何用硬件实现 SM 图。接着我们介绍了微程序方法，并讨论了怎样针对不同类型的微程序对 SM 图进行转换。我们还介绍了怎样用有链接关系的状态机来表述一个复杂系统分解为几个简单系统。本章中骰子模拟机的设计就体现了这一技术。

第 6 章给出了有关基于 FPGA 实现数字系统的一些问题。为了说明对 FPGA 的映射过程，首先我们通过几个简单的例子对 FPGA 的组成模块进行了手工映射。本章中还介绍了如何使用香农公式把具有较多变量的表达式分解为几个具有较少变量的表达式。同时，还介绍了现代 FPGA 的特点，如进位链、级联链、专用存储器和专用乘法器等。我们没有在特定的商用器件上介绍 FPGA 的这些特点，而是在通用结构上加以说明。学生们一旦理解了一般原理，则他们就可以理解要用的任何具体的商用器件。本章中也介绍了软件设计流程中的过程和算法，简单介绍了综合映射、布局和路由的过程，并且说明了综合中的优化过程。

第 7 章介绍了浮点数运算的基本技术。本章中首先介绍了二进制补码浮点数格式，接着介绍了 IEEE 标准浮点数格式，随后通过基本算法的展开介绍了浮点数的乘法例子，然后用 VHDL 对此系统进行仿真，最后综合并用 FPGA 实现。有些教师可能喜欢在介绍第 7 章前先介绍第 8 章和第 9 章，即使不介绍第 7 章也不会影响到授课的连续性。

当学生们学到第 8 章的时候，他们对于 VHDL 语言的基本部分已经很熟悉了。所以本章介绍了一些 VHDL 的更高级的功能和应用。作为很重要的讨论议题之一，我们介绍了包含 IEEE-1164 标准逻辑在内的多值逻辑。随后我们用带有三态输出总线的存储器模型来说明多值逻辑的使用。

第 9 章介绍了微处理器设计。首先介绍了指令集合构造（ISA）。处理器为早期 RISC 处理器：MIPS R2000。处理器 MIPS 的重要指令均加以介绍后，我们设计实现了其部分指令。同时，对处理器各个组成部分的设计，例如指令存储模块、数据存储模块和寄存器文件等逐一分别介绍。接着把这些组成模块集成在一起，就形成了一个完整的处理器。这一模块可以在测试平台上进行验证或者可以在 FPGA 上进行综合实现。为了在 FPGA 上对设计进行测试，我们需要写出设计模块的输入–输出模块。为了理解本例子，需要了解汇编语言编程和计算机组成的基本原理。

第 10 章包含了硬件测试和可测性设计中的一些重要议题。首先介绍了组合逻辑和时序逻辑的测试技术，随后介绍了扫描设计和边界扫描技术，此技术为数字系统测试提供了方便。最后对内置自测（BIST）进行了讨论，给出了边界扫描和 BIST 的 VHDL 程序代码例子。在数字系统设计中，本章内容起很重要的作用，我们建议在与此课程相关的所有课程中都介绍本章的内容。第 10 章的内容在第 8 章讲解完之后就可以进行讨论了。

第 11 章给出了三个完整的设计实例，用于说明 VHDL 综合工具的使用方法。首先，手表设计实例展示了从文本设计到状态图，最后到 VHDL 描述的具体设计流程。本实例是一个模块化设计。随后，手表设计的测试程序展示了如何使用多种过程调用实现设计的测试。第二个实例是使用 VHDL 程序实现一个 RAM 存储器。第三个实例是一个串口通信收发机，对于已经学习完第 8 章的同学来说，此实例很容易理解。

为那些使用过本书第一版的教师，这里给出了一个映射图，以便了解第二版和第一版的不同之处。在本书中，我们不再使用 BITLIB 库文件，而是先使用 IEEE numeric-bit 库文件直到第 8 章引入了多值逻辑，之后就用 IEEE numeric-std 多值逻辑库文件。所有程序中均使用 IEEE 的标

准库文件，而不是使用 BITLIB 库文件。

第 1 章	增加了简单的 Mealy 和 Moore 设计，而在时序电路时序章节里增加了更为详尽的介绍
第 2 章	增加了设计流程和设计技术的概述。第一版第 2 章中的函数和过程语句移到第二版的第 8 章。第一版第 8 章中的惯性延迟和传输延迟移到第二版的第 2 章。第二章就开始介绍综合，而本书中所有给出的代码均可综合
第 3 章	保留了第一版中的开始几节，增加了 CPLD 和 FPGA 的新内容。第一版本第三章中的设计例子（交通灯、键盘扫描器）移到第四章
第 4 章	增加了多个新设计实例。对第一版第 4 章中的例子重新整合，使其变为可综合的代码。第一版第 3 章中的两个实例移到本章
第 5 章	增加了有关微程序的更详尽的内容
第 6 章	新增了关于 FPGA 的一般原理，没有局限于某个具体的商用器件，而是作为例子引出典型的几个商用器件；增加了软件设计流程的简单介绍，包括映射、布局和布线的原理
第 7 章	增加了 IEEE 标准浮点数格式和浮点数加法器设计例子
第 8 章	第一版第 2 章中的函数和过程语句移到本章。第一版第 8 章的大部分章节保留下来，第一版第 9 章中的内存模块移到本章作为多值逻辑设计实例
第 9 章	这是新增加的一章。本章中介绍了 MIPS 指令集和 MIPS 处理器的设计。第一版第 9 章中的内存模块移到第二版第 8 章和第 11 章，并且去掉了第一版第 9 章中的总线模块
第 10 章	增加了对边界扫描和 STUMPS 结构的详尽描述
第 11 章	增加了新的设计实例（手表）。第一版第 9 章中的存储器时序模块出现在本章。重新整合了第一版第 11 章的 UART 设计，同时删掉了微控制器的设计

本书是作者多年来在得克萨斯大学奥斯汀分校讲授高年级数字系统设计高级课程经验累积而成的。这些年来，数字系统的硬件实现一直在不断地变化，但是许多相同的设计原理仍然在应用。在本课程的开始几年，我们手画模型，包括具体的晶体管到实现我们的设计。然后，引入了集成电路，我们可以采用面包板和 TTL 电路来实现我们的设计。现在，我们可以用 FPGA 和 CPLD 来实现非常复杂的设计。我们最初在大型机上用硬件描述语言和仿真器。当出现 PC 机时，我们写改进的硬件描述语言，并且在 PC 机上进行模拟仿真。当 VHDL 作为 IEEE 的标准语言在业界广泛应用后，我转向了 VHDL。当今高性能的商业 CAD 工具的广泛应用，使我们能够直接从 VHDL 程序代码综合实现复杂的设计。

本书中的所有 VHDL 代码都使用 ModelSim 仿真工具进行了验证。ModelSim 软件的学生版是可用的，所以我们建议与本书配合使用。通过本书的配套 CD 可以下载 ModelSim 的学生版，并且 CD 中介绍了如何使用此软件。CD 中有两个软件包：LogicAid 和 SimUaid。这两个软件对数字系统设计的教学都很有帮助，同时 CD 中还包括软件的使用手册和实例。

致　　谢

我们对为本书付出时间和精力的所有人表示由衷的感谢。多年来，本书收到了学习我们数字系统设计课程同学们的宝贵反馈意见。我们特别感谢为本书的第一版评阅和本书的改进提出许多建议的教师们。这些教师包括：

Gang Feng，普拉特河威斯康星大学

Elmer. A. Grubbs，亚利桑那大学

Marius Z. Jankowski，南缅因大学

Chun-Shin Lin，密苏里–哥伦比亚大学

Peter N. Marinos，杜克大学

Maryam Moussavi，长滩加利福尼亚州立大学

Aaron Striegel，圣母大学

Peixin Zhong，密歇根州立大学

特别感谢 Mentor Graphics 公司的 Ian Burgess 先生为本书配备 ModelSim 学生版软件。我们也对 Chris Carson, Hilda Gowans, Rose Kernan 和 Kamilah Reid Burrell 在出版过程中给予的帮助表示感谢，并非常高兴与他们一同工作。我们也借此机会对那些学生在文字处理、VHDL 代码测试、CD 和画图方面给予的帮助表示感谢，这些学生包括 Ciji Isen, Roger Chen, William Earle, Manish Kapadia, Matt Morgan, Elizabeth Norris 和 Raman Suri。

Charles. H. Roth, Jr.

Lizy K. John



目　录

第 1 章　逻辑设计基本原理简介

这一章节将回顾许多逻辑设计的一些基本原理，这些内容一般在逻辑设计初级课程中讲授，也会提到一些将在以后的章节里引用的例子。对本章节里讨论的内容如果想多了解一些，读者可查阅经典的逻辑设计基础教程，例如《逻辑设计基础》第五版（Roth, *Fundamentals of Logic Design, 5th Edition*. Thomson Brooks/Cole, 2004）。首先，我们复习组合逻辑电路，随后复习时序逻辑。组合逻辑中没有存储单元，所以其当前输出只与当前输入有关。时序逻辑中有存储单元，所以其当前的输出不仅与当前的输入有关而且与以前的输入也有关。本章中介绍的时序电路时序和同步设计的内容很重要，只有熟练掌握了时序的相关内容，才能更好地进行数字系统设计。

1.1　组合逻辑电路

图 1.1 中列举了一些逻辑电路中常用的基本逻辑门符号。如果没有另外说明，我们指定逻辑变量取值为 0 和 1 两种。通常，我们使用正逻辑，即低电平对应逻辑 0，高电平对应逻辑 1。相对地，当使用负逻辑时，低电平对应逻辑 1，高电平对应逻辑 0。

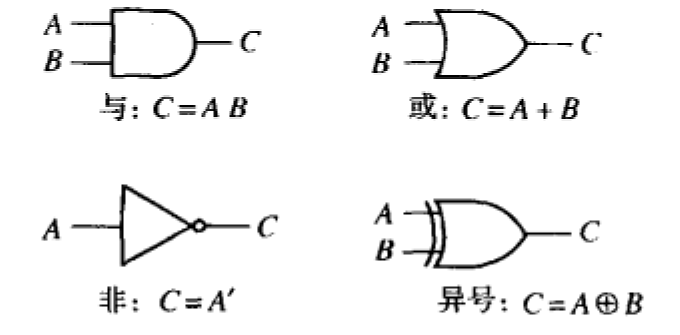


图 1.1　基本逻辑门

对图 1.1 所示的与门（AND）来说，当且仅当输入 $A=1$ 且 $B=1$ 时输出 $C=1$。我们使用点乘(·)或将变量并排写出来表示逻辑与（AND）运算，即 $C=A\ \text{AND}\ B=A\cdot B=AB$。

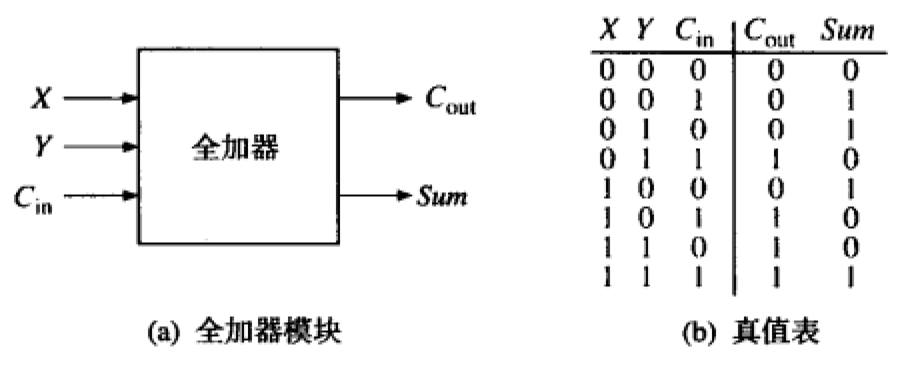
对或门（OR）来说，当且仅当输入 $A=1$ 或 $B=1$ 时，输出 $C=1$。我们使用符号+表示或（OR）运算，即 $C=A\ \text{OR}\ B=A+B$。

非门（NOT）也叫做反相器，它的作用是对输入量取反。例如，当 $A=1$ 时，$C=0$，而当 $A=0$ 时，$C=1$。我们使用符号'表示取反运算，即 $C=\text{NOT}\ A=A'$。

对异或门（XOR）来说，当输入 $A=1,B=0$ 或 $A=0,B=1$ 时，输出 $C=1$。我们使用符号 ⊕ 表示异或运算，因此可以写为

$$C=A\ \text{XOR}\ B=AB'+A'B=A\oplus B \tag{1.1}$$

一个组合逻辑电路的功能可以用真值表来描述。真值表中每组输入值都对应一个输出值。例如全加器（参见图 1.2），其功能为计算两个二进制数（X 和 Y）与前级进位（C_{in}）的和（Sum）并给出下级进位（C_{out}）。例如，当输入 $X=0, Y=0$，$C_{\text{in}}=1$ 时，三个输入的和为 $0+0+1=01$，因此输出 $Sum=1$，$C_{\text{out}}=0$。当输入 $X=0, Y=1$，$C_{\text{in}}=1$ 时，$0+1+1=10$，输出 $Sum=0$，$C_{\text{out}}=1$。当输入 $X=Y=C_{\text{in}}=1$ 时，1+1+1=11，输出 $Sum=1$，$C_{\text{out}}=1$。



X	Y	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) 全加器模块　　(b) 真值表

图 1.2　全加器

我们用真值表来得到 *Sum* 和 C_{out} 的代数表达式。由真值表可得，当 $X=0, Y=0, C_{in}=1$ 时，$X'Y'C_{in}=1$；当 $X=0, Y=1, C_{in}=0$ 时，$X'YC'_{in}=1$；当 $X=Y=C_{in}=1$ 时，$XYC_{in}=1$。由此可得 *Sum* 的逻辑表达式为上述四项的和（或）:

$$Sum = X'Y'C_{in} + X'YC'_{in} + XY'C_{in} + XYC_{in} \tag{1.2}$$

对于给定的输入，上面与或和的四项中只能有一项为 1。同理，C_{out} 的表达式也可以用四项的或来表示：

$$C_{out} = X'YC_{in} + XY'C_{in} + XYC'_{in} + XYC_{in} \tag{1.3}$$

式(1.2)和式(1.3)中的各项均为最小项，上述两式的表示方式为最小项和的形式。我们也可以采用多进制或十进制把上面两个表达式记为

$$Sum = m_1 + m_2 + m_4 + m_7 = \sum m(1,2,4,7)$$
$$C_{out} = m_3 + m_5 + m_6 + m_7 = \sum m(3,5,6,7)$$

上面的十进制数代表真值表中使逻辑表达式为 1 的最小项所在的行。例如在图 1.2(b)中，若 $Sum=1$，我们可以看出对应行 001, 010, 100, 111（行 1, 2, 4, 7）。

逻辑函数表达式也可以用使函数等于 0 时的各项输入的组合来表示。在全加器的真值表中，当 $X=Y=C_{in}=0$ 时，$C_{out}=0$。即 $C_{out}=0$ 时，$X+Y+C_{in}$ 项才等于 0。同样，当 $X=Y=0, C_{in}=1$ 时 $X+Y+C'_{in}$ 等于 0；当 $X=C_{in}=0, Y=1$ 时，$X+Y'+C_{in}$ 等于 0；当 $Y=C_{in}=0, X=1$ 时，$X'+Y+C_{in}$ 等于 0。由此可得 C_{out} 的逻辑表达式为上述四项的与：

$$C_{out} = (X+Y+C_{in})(X+Y+C'_{in})(X+Y'+C_{in})(X'+Y+C_{in}) \tag{1.4}$$

只有当真值表中行输入对应 000, 001, 010, 100 时，C_{out} 值为 0，此时对应真值表第 0, 1, 2, 4 行。而对于真值表中的其他行，则必有输出 C_{out} 值为 1。式(1.4)中的每一项都称为最大项，式(1.4)也称为最大项表示法。同样，最大项表述也可用十进制表示如下：

$$C_{out} = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \Pi M(0,1,2,4)$$

其中，十进制数对应 C_{out} 值为 0 时真值表中的行数。

1.2　布尔代数与代数式的化简

逻辑设计中用到的基础数学知识就是布尔代数。表 1.1 总结了布尔代数的定律和定理，并以对偶对的方式列出。例如，式(1.10D)与式(1.10)是对偶的。对于二值逻辑，布尔代数可以很容易地通过真值表进行检验。运用这些定律和定理，可以将逻辑表达式化简。

摩根定律是布尔代数中一个很重要的定律。使用式(1.16)和式(1.16D)给出的摩根定律，可以

通过一步步的推导最终得到表达式的互补形式。摩根定律的一般形式(1.17)式子，可以用来直接一步得到表达式的互补形式。式(1.17)可以解释为：要得到一个布尔表达式的互补式，先对每个变量取反，再用 0 代替 1，用 1 代替 0，与变或，或变与。为保证运算先后层次分明，最好加括号。如果函数 F 中，先与后或，则在 F'中，要加括号来保证先或后与。

例如，要找出函数 F 的互补式，则有

$$F = X + E'K(C(AB + D')\cdot 1 + WZ'(G'H + 0))$$
$$F' = X'(E + K' + (C' + (A' + B')D + 0)(W' + Z + (G + H')\cdot 1))$$

当 F 中 AND 操作被变为 OR 操作，F'中对应部分就要加括号。求一个表达式的对偶式与此类似，但是表达式中变量不取反。

表 1.1　布尔代数定律和定理

与 0 和 1 的运算：			
$X + 0 = X$	(1.5)	$X \cdot 1 = X$	(1.5D)
$X + 1 = 1$	(1.6)	$X \cdot 0 = 0$	(1.6D)
重叠律：			
$X + X = X$	(1.7)	$X \cdot X = X$	(1.7D)
还原律：			
$(X')' = X$	(1.8)		
互补律：			
$X + X' = 1$	(1.9)	$X \cdot X' = 0$	(1.9D)
交换律：			
$X + Y = Y + X$	(1.10)	$X \cdot Y = Y \cdot X$	(1.10D)
结合律：			
$(X + Y) + Z = X + (Y + Z)$ $= X + Y + Z$	(1.11)	$(XY)Z = X(YZ) = XYZ$	(1.11D)
分配律：			
$X(Y + Z) = XY + XZ$	(1.12)	$X + YZ = (X + Y)(X + Z)$	(1.12D)
化简定理：			
$XY + XY' = X$	(1.13)	$(X + Y)(X + Y') = X$	(1.13D)
$X + XY = X$	(1.14)	$X(X + Y) = X$	(1.14D)
$(X + Y')Y = XY$	(1.15)	$XY' + Y = X + Y$	(1.15D)
摩根定律：			
$(X + Y + Z + \cdots)' = X'Y'Z'\cdots$	(1.16)	$(XYZ\cdots)' = X' + Y' + Z' + \cdots$	(1.16D)
$[f(X_1, X_2, \cdots, X_n, 0, 1, +, \cdot)]' = f(X_1, X_2, \cdots, X_n, 0, 1, \cdot, +)$			(1.17)
对偶式：			
$(X + Y + Z + \cdots)^D = XYZ\cdots$	(1.18)	$(XYZ\cdots)^D = X + Y + Z + \cdots$	(1.18D)
$[f(X_1, X_2, \cdots, X_n, 0, 1, +, \cdots)]^D = f(X_1, X_2, \cdots, X_n, 0, 1, \cdots, +)$			(1.19)
展开和因式分解定理：			
$(X + Y)(X' + Z) = XZ + X'Y$	(1.20)	$XY + X'Z = (X + Z)(X' + Y)$	(1.20D)
蕴涵定理：			
$XY + YZ + X'Z = XY + X'Z$	(1.21)	$(X + Y)(Y + Z)(X' + Z)$ $= (X + Y)(X' + Z)$	(1.21D)

用表 1.1 给出的定律定理，化简逻辑表达式的 4 个步骤如下：

1. 合并项。

使用定理 $XY+XY'=X$ 合并两个项，例如，

$$ABC'D'+ABCD'=ABD'[X=ABD',Y=C]$$

采用这种方法合并项时，要合并的两个项必须含有相同的变量，并且其中必有一变量在一项中取补，另一项中不取补。由于 $X+X=X$，对于重复出现的项，可只保留一项。例如，对 C_{out} [如式(1.3)所示]，通过合并第 1 项和第 4 项、第 2 项和第 4 项、第 3 项和第 4 项，可以化简如下：

$$\begin{aligned} C_{out} &= (X'YC_{in}+XYC_{in})+(XY'C_{in}+XYC_{in})+(XYC'_{in}+XYC_{in}) \\ &= YC_{in}+XC_{in}+XY \end{aligned} \tag{1.22}$$

注意，式(1.3)中的第 4 项用了三次。

当 X, Y 表达式更复杂时，我们可以使用定理进行化简。例如，

$$(A+BC)(D+E')+A'(B'+C')(D+E')=D+E'$$
$$[X=D+E',Y=A+BC,Y'=A'(B'+C')]$$

2. 消除项。

在可能的情况下，使用定理 $X+XY=X$ 来消除冗余项，接着应用蕴涵定理($XY+X'Z+YZ=XY+X'Z$)去消除蕴涵项，例如，

$$A'B+A'BC=A'B[X=A'B]$$
$$A'BC'+BCD+A'BD=A'BC'+BCD\ [X=C,Y=BD,Z=A'B]$$

3. 消除因子。

应用定理 $X+X'Y=X+Y$ 去消除冗余的因子。在应用此定理之前，有必要先进行简单因式分解。例如，

$$\begin{aligned} & A'B+A'B'C'D'+ABCD' \\ &= A'(B+B'C'D')+ABCD' && \text{[根据式(1.12)]} \\ &= A'(B+C'D')+ABCD' && \text{[根据式(1.15D)]} \\ &= B(A'+ACD')+A'C'D' && \text{[根据式(1.10)]} \\ &= B(A'+CD')+A'C'D' && \text{[根据式(1.15D)]} \\ &= A'B+BCD'+A'C'D' && \text{[根据式(1.12)]} \end{aligned}$$

通过上述三种方法得到的表达式未必是最简的表达式。如果无法用上述三种方法继续化简，则可以引入一个新的冗余项来帮助进一步化简。

4. 添加冗余项。

有好几种方法可以引入冗余项，比如加上 XX'，乘以($X+X'$)，给 $XY+X'Z$ 加上 YZ（蕴涵定理），或者给 X 加上 XY。如果有可能的话，应该选择这样的添加项使得它们可以和其他项合并或消除其他项。例如，

$$\begin{aligned} & WX+XY+X'Z'+WY'Z' \text{（加入 } WZ'\text{）} \\ &= WX+XY+X'Z'+WY'Z'+WZ' \text{（消去 } WY'Z'\text{）} \\ &= WX+XY+X'Z'+WZ' \text{（消去 } WZ'\text{）} \end{aligned}$$

$$= WX + XY + X'Z'$$

当展开或分解一个函数表达式时，除了常用的分配律(1.12)之外，第二个分配律(1.12D)和定理(1.20)非常有用。下面的例子就是将和之积式变为积之和式。

$$(A + B + D)(A + B' + C)(A' + B + D')(A' + B + C')$$
$$= (A + (B + D)(B' + C'))(A' + B + C'D') \quad [式(1.12D)]$$
$$= (A + BC' + B'D)(A' + B + C'D') \quad [式(1.20)]$$
$$= A(B + C'D') + A'(BC' + B'D) \quad [式(1.20)]$$
$$= AB + AC'D' + A'BC' + A'B'D \quad [式(1.12)]$$

应该注意到，在使用通常的分配律(1.12)之前，已经先用第二个分配律(1.12D)和定理（1.20）进行了化简。任意布尔代数式都可以通过两个分配律（1.12 和 1.12D）和定理(1.20D)来分解因式。作为因式分解的例子，可以从后向前顺序来分析上面的例子即可。

下面是异或运算法则：

$$X \oplus 0 = X \quad (1.23)$$
$$X \oplus 1 = X' \quad (1.24)$$
$$X \oplus X = 0 \quad (1.25)$$
$$X \oplus X' = 1 \quad (1.26)$$
$$X \oplus Y = Y \oplus X \quad （交换律） \quad (1.27)$$
$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z \quad （结合律） \quad (1.28)$$
$$X(Y \oplus Z) = XY \oplus XZ \quad （分配律） \quad (1.29)$$
$$(X \oplus Y)' = X \oplus Y' = X' \oplus Y = XY + X'Y' \quad (1.30)$$

利用式(1.1)和式(1.30)，可以将 (1.2)式的 *Sum* 表达式可以用异或重新写为

$$Sum = X'(Y'C_{in} + YC'_{in}) + X \oplus Y'C'_{in} + YC_{in}$$
$$= X'(Y \oplus C_{in}) + X(Y \oplus C_{in})' = X \oplus Y \oplus C_{in} \quad (1.31)$$

当电路优化为最少门数时候，本章中的化简规则是很重要的。另外，当把电路映射到某一特定目标器件（例如，只有 NAND 和 NOR 门可以使用）的时候，表达式的等价形式是很有用的。

1.3　卡诺图

用卡诺图提供了化简三到五个变量逻辑函数的一个简便的手段。图 1.3 是一个四变量卡诺图。图中每个方格表示 16 个可能的四变量最小项中的一个。图中方格内的数字 1 表明函数中有相应的最小项，而 0（或空白）表明没有相应的最小项，而格内的 X 则表示我们并不在意该最小项的存在与否（称为随意项）。随意项的出现有两种情况：(1)与随意项对应的输入变量取值组合永不出现；(2)虽然输入变量取值组合出现，但是电路的输出与此无关。

卡诺图中的横向和纵向变量的取值排序是有规律的，保证了图中相邻小方格只有一个变量取值不同。图中第一列和最后一列、第一行和最后一行可以看成是相邻的。标有 1 的两个相邻小方格可以基于法则 $xy + xy' = x$ 去掉一个变量后合并为一个。图 1.3 表示有九个最小项和两个随意项的四变量函数卡诺图。最小项 $A'BC'D$ 和 $A'BCD$ 只有变量 C 不同，因此它们可以合并为 $A'BD$，如图用圈所示。若有四个 1 的分布呈对称，则可以合并消去两个变量。图中四个角落的 1，如图 1.3(b)所示的圈可以合并如下：



$$(A'B'C'D' + AB'C'D') + (A'B'CD' + AB'CD') = B'C'D' + B'CD' = B'D'$$

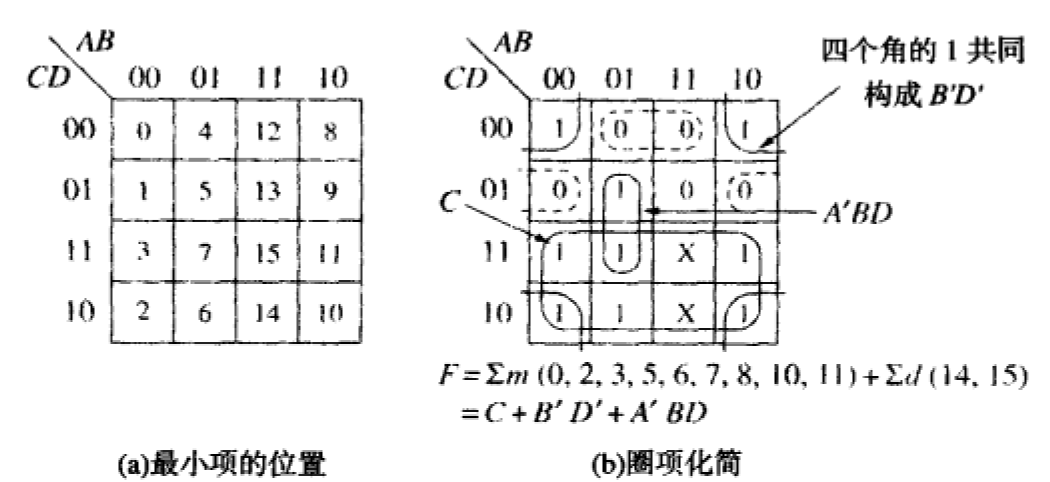


图 1.3　四变量卡诺图

同样，图中下半部的六个 1 和两个 X 也可以合并消除三个变量后成为 C。这样最后化简的结果为

$$F = A'BD + B'D' + C$$

一个函数的最小与或表达式由素（质）蕴涵项构成。卡诺图中的一个、两个、四个或八个相邻的 1 圈项而得到的蕴涵项，如果不能再与其他 1 的圈项而得到的蕴涵项合并消去变量，则它就是素（质）蕴涵项。如果一个素（质）蕴涵项所圈的小方格中至少有一个不被其他素（质）蕴涵项所圈，那么它就称为基本素（质）蕴涵项。在求函数的最小与或表达式时，应首先圈出首要基本素（质）蕴涵项，然后用最少的圈圈出余下 1。图 1.4 所示的卡诺图有 5 个蕴涵项和 3 个基本素（质）蕴涵项。$A'C'$是一个基本素（质）蕴涵项，因为 m_1 没有被其他素（质）蕴涵项所覆盖。由于 m_{11} 和 m_2,ACD 和 $A'B'D'$也都是一个基本素（质）蕴涵项。圈出了基本素（质）蕴涵项后，除了 m_7 项所有的 1 都被覆盖了。由于 m_7 可以包括在 $A'BD$ 或 BCD 中，因此，F 有两种最小化形式：

$$F' = A'C' + A'B'D + ACD + A'BD$$

和

$$F = A'C + A'B'D + ACD + BCD$$

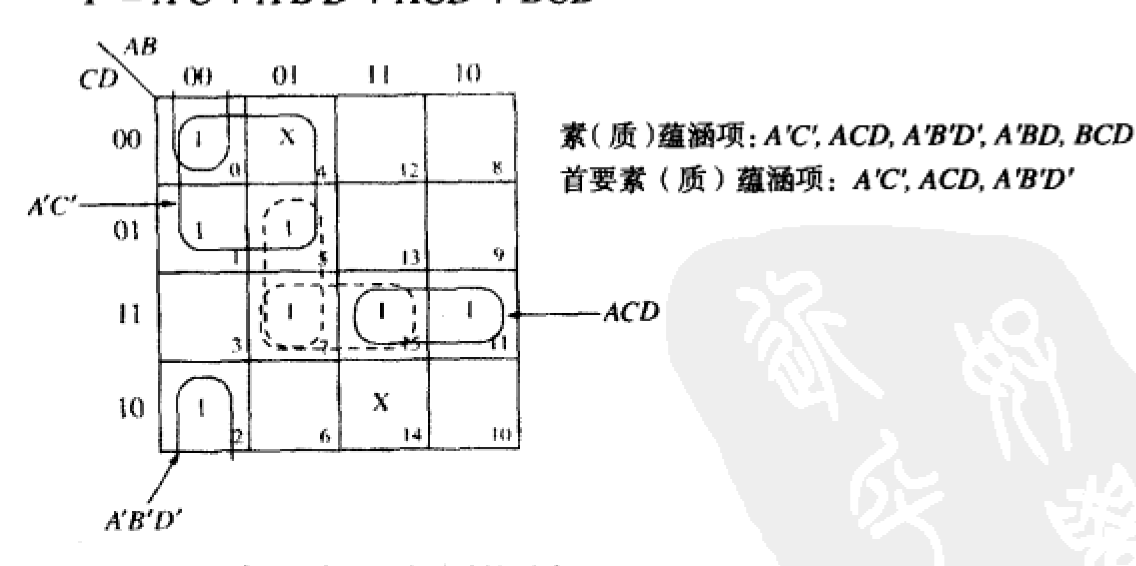


图 1.4　素（质）蕴涵项的选择

当图中存在随意项 X 时，如果要生成素（质）蕴涵项，则置 X 为 1；但如果要寻找覆盖所有 1 的素（质）蕴涵项的最小集合时，则可以忽略 X。用以下步骤可以从卡诺图中获得函数的最小与或表达式：

1. 选择一个还没有圈过的最小项（一个 1）。
2. 找出所有与该最小项相邻的 1 和 X（检查在 n 变量卡诺图上的 n 个相邻的方格）。
3. 若有一项能覆盖该最小项和所有与之相邻的 1 和 X，那么该项就是基本素（质）蕴涵项，于是选定该项（注意，在步骤 2 和步骤 3 中随意项可以视为 1，而在步骤 1 中却不可）。
4. 重复步骤 1, 2, 3，直到所有基本素（质）蕴涵项都被找出。
5. 查找一个覆盖图上所有剩余 1 的素（质）蕴涵项的最小集合（如果有多个这样的集合，则选择包含变量数目最少的集合）。

为了从卡诺图中得到最小或与表达式，则通过圈 0 来实现。因为 F 中的 0 就是 F' 中的 1，所以通过圈 0 可以得到 F' 的最小与或表达式，取补即是 F 的最小或与表达式。对于图 1.3，首先可以圈出 F' 的基本素（质）蕴涵项（$BC'D'$ 和 $B'C'D$，如虚线所圈），并且用 AB 圈出所剩的 0。这样得到 F'的最小与或表达式为

$$F' = BC'D' + B'C'D + ABC'$$

由此，得 F 的最小或与表达式：

$$F = (B' + C + D)(B + C + D')(A' + B' + C)$$

1.3.1 用卡诺图中嵌入的变量进行化简

两个四变量卡诺图可以用于化简五变量函数表达式。如果表达式的变量个数大于 5，则可以在卡诺图中嵌入变量进行化简。如表 1.2 所示，真值表有 6 个输入变量（A, B, C, D, E, F）和 1 个输出变量（G）。真值表中只给出了部分取值，而对应于输出为 0 的所有输入取值没有给出。如果要列出这一真值表的全部取值，则需要 64 行。。

表 1.2　一个六变量表达式的部分真值表

A	B	C	D	E	F	G
0	0	0	0	X	X	1
0	0	0	1	X	X	X
0	0	1	0	X	X	1
0	0	1	1	X	X	1
0	1	0	1	1	X	1
0	1	1	1	1	X	1
1	0	0	1	X	1	1
1	0	1	0	X	X	X
1	0	1	1	X	X	1
1	1	0	1	X	X	X
1	1	1	1	X	X	1

通过嵌入变量的方法，卡诺图可以扩展用于化简多变量函数。由于输入变量 E 和 F 的大部分值为随意项（X），所以我们可以构建一个四变量卡诺图，使输入变量为 A, B, C, D，而剩下的两个输入 E 和 F 变量嵌入到卡诺图中。图 1.5 给出的是在小方格中嵌入两个变量 E 和 F 的四变量卡诺图。当小格中标有 E 时，如果 $E = 1$，表示函数 G 里存在相应的最小项，如果 $E = 0$，则表示没有相应的最小项。真值表中第五行和第六行的取值决定了 E 在卡诺图中出现的方格，对应了最小项 m_5 和 m_7，而第七行的取值决定了 F 在卡诺图中出现方格，对应于 m_9。这样该卡诺图可以表示六变量函数：

$$G(A, B, C, D, E, F) = m_0 + m_2 + m_3 + Em_5 + Em_7 + Fm_9 + m_{11} + m_{15} \text{（+ 随意项）}$$

其中，最小项均由变量 A, B, C, D 组成。注意，只有当 $F = 1$ 时，G 中才有 m_9 项。

下面我们来讨论用嵌入变量卡诺图变量化简函数的一般方法。一般说来，如果函数 F 的卡诺图中小格 m_j 内有变量 P_i，则表示当 $P_i = 1$ 且 $m_j = 1$ 时，$F = 1$。若卡诺图嵌入了变量 $P_1, P_2, \cdots$，则 F 的最小与或表达式可以通过寻找一个形如

$$F = MS_0 + P_1MS_1 + P_2MS_2 + \cdots \tag{1.32}$$

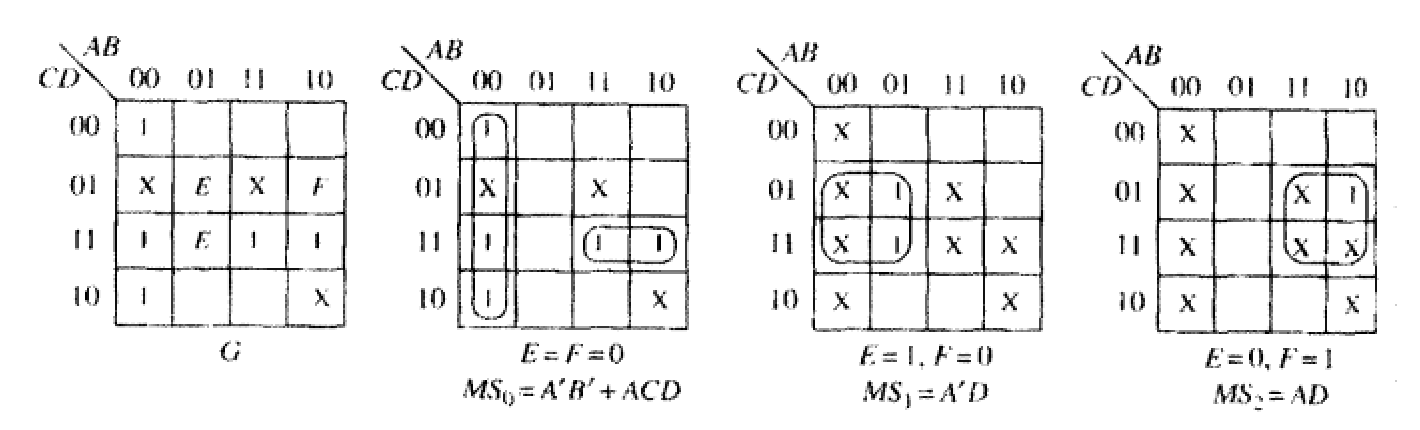


图 1.5　嵌入卡诺图变量化简

的与或表达式得到，其中，

- MS_0 是 $P_1 = P_2 = \cdots = 0$ 时对应的最小与或表达式。
- MS_1 是 $P_1 = 1$，$P_j = 0\ (j \neq 1)$，并且将所有 1 变为随意项后对应的最小与或表达式。
- MS_2 是 $P_2 = 1$，$P_j = 0\ (j \neq 2)$，并且将所有 1 变为随意项后对应的最小与或表达式。

对于卡诺图中其他嵌入变量，均可以参照上述方法得到对应的最小与或表达式。

函数 F 的这个最终表达式始终是 F 的一个正确表达式。如果所嵌入变量的赋值是独立的，则 F 的这个表达式就是最小化与或表达式。不然，若嵌入变量彼此不独立（如 $P_1=P'_2$），则 F 的这个表示就不是最小化的与或表达式。

对于图 1.5 的例子，找 MS_0，MS_1 和 MS_2 的过程如图所示，其中 E 对应 P_1，F 对应 P_2。所得结果就是函数 G 的最小与或表达式

$$G = A'B' + ACD + EA'D + FAD$$

经过多次练习，就可以从原卡诺图上直接写出每一个变量对应的最小与或表达式，而不用另画出每一个变量对应的卡诺图再求其最小与或表达式。

1.4　用与非门和或非门进行设计

在许多技术中，与非门和或非门比用与门和或门更容易实现。与非门和或非门的逻辑符号如图 1.6 所示，其中输入端和输出端的小圆圈表示取反。任何逻辑函数都可以只用与非门或者只用或非门来实现。

与非门

$C=(AB)'=A'+B'$

或非门

$C=(A+B)'=A'B'$

图 1.6　与非门和或非门

与门和或门组成的逻辑电路可以直接转换为只用或非门（或者与非门）的逻辑电路。设计或非门组成的电路，首先要得到函数的或与式（卡诺图中圈 0），然后找到输出端为与门的或门和与门电路；如果与门输出不对应与门输入，或门输出不对应或门输入，那么即把全部的门转换为或非门，并且必要时把输入取反。图 1.7 举例说明对下式的转换过程：

$$Z = G(E + F)(A + B' + D)(C + D) = G(E + F)[(A + B')C + D]$$

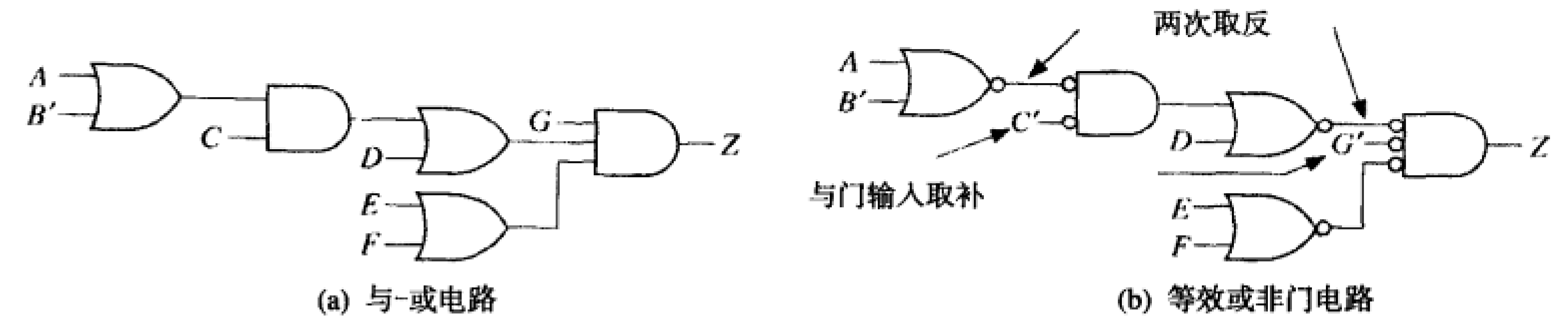


图 1.7　或非门的转换

我们可以通过类似的过程把电路转换为与非门电路。不同之处就在于，首先我们要得到函数的与或表达式（卡诺图中圈 1），且电路的最后输出门应为或门。

即使电路中与门和或门不能相互代替，我们仍能将与或电路转换为或非门电路或者与非电路，但可能需要增加额外的反相器，使之附加的反相运算被另一个反相运算抵消。我们可采用以下步骤得到与非门（或非门）电路：

1. 通过在输出端加入一个小圆圈（简称取反圈），把所有与门改为与非门。通过在输入端加入一个取反圈，把所有或门换改为与非门（如果想得到或非门电路，则在所有或门输出端和与门输入端均加入一个取反圈即可）。
2. 若反相输出与反相输入相连，则不需要任何操作，因为两次取反可以互相抵消。
3. 若非反相输出和反相输入相连（或者反过来），则需要加入一个非门以消除取反圈（根据需要，在输入端或输出端放入带有取反圈的非门）。
4. 若一个变量连接到一个反相输入端，则对该变量取反（或增加非门），以抵消输入端的取反运算。

换言之，如果我们总是成对添加取反圈（或者反相器），则电路所实现的函数功能将不会改变。图1.8(a)举例说明如何把一个与门和或门电路转换为与非门电路。首先，我们添加反相器，将全部的与门改为与非门[参见图 1.8(b)]。四处粗线表示只增加了一个取反运算。为此，可以通过添加两个反相器和对两变量取补，就可以修正弥补，如图 1.8(c)所示。

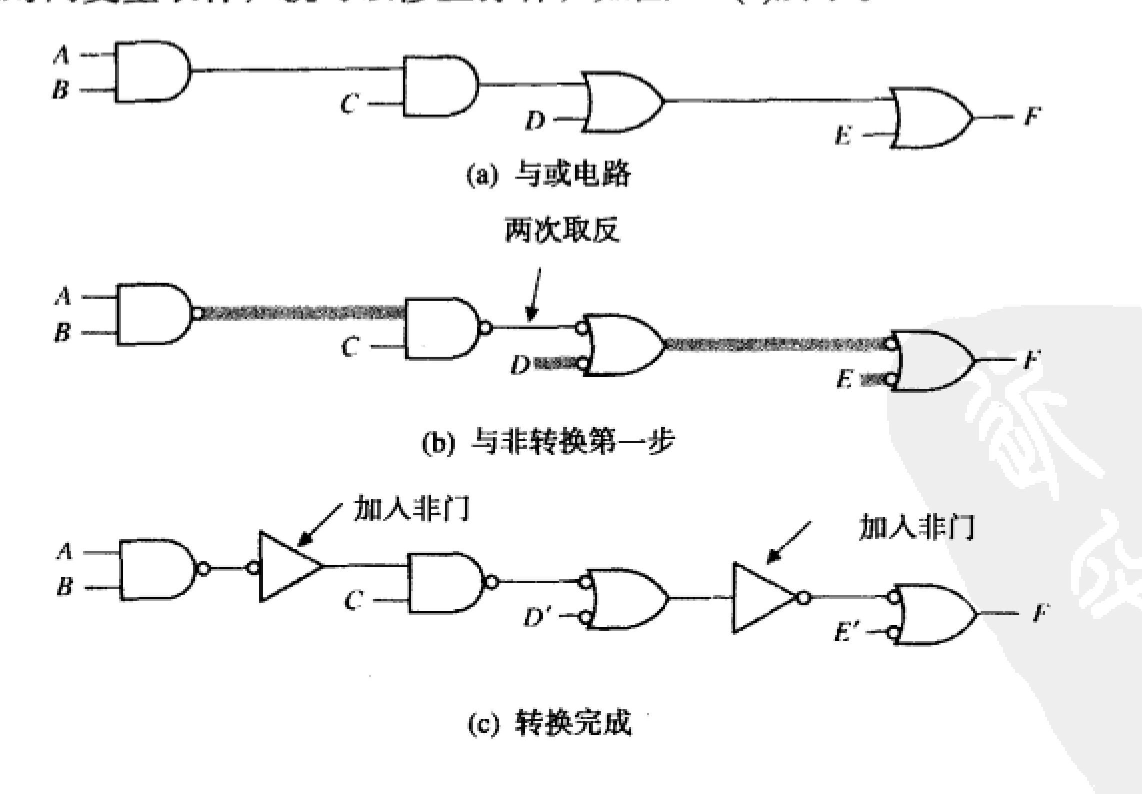
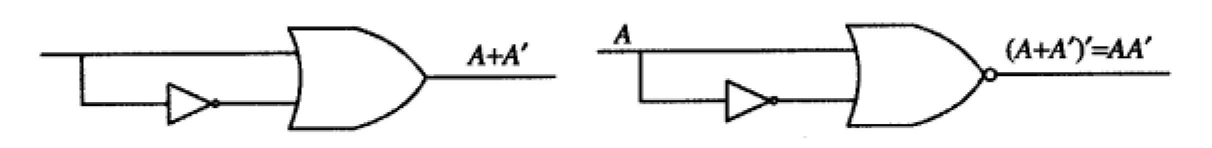


图 1.8　把与-或门电路转换为与非门电路

1.5 组合电路中的冒险

当组合电路的输入改变时，输出端有可能出现不希望的瞬间波动。这种瞬间波动是由输入到输出的不同路径的不同传输延时所致，对应于输入的一个变化，由于一些传输延迟的组合，电路响应本应该恒为 1 的输出，如果出现瞬间的 0，则我们称此电路具有一个静态 1 冒险。同样，如果输出响应本应该恒为 0 的输出，如果出现瞬间的 1，则我们称此电路存在一个静态 0 冒险。当输出响应变化（由 0 变为 1，或者由 1 变为 0）时，如果发生三次或三次以上的瞬时变化，则我们称此电路存在一个动态冒险。

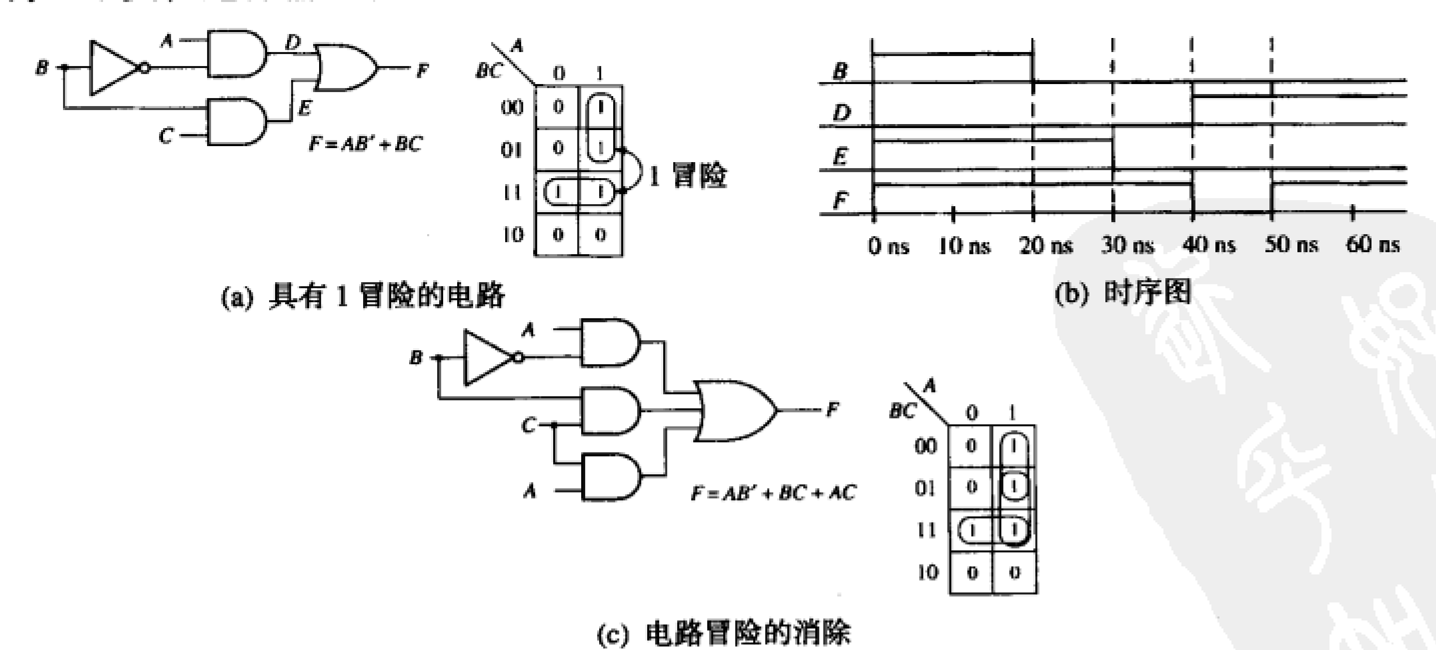
观察图 1.9 中两个简单的电路。当使用一个反相器和一个或门实现表达式 $A + A'$时[参见图 1.9(a)]，此电路的输出本应该恒为 1，但是反相器的延时会使该电路存在静态冒险。假设反相器具有非零延时且输入 A 恰好从 1 变为 0 时，则信号经过反相器延迟一小段时间间隔，此段时间内或门的输入却均为 0，因此此时电路的输出就会瞬变为 0。相同对于如图 1.9(b)所示的电路，其输出应恒为 0。但是，当输入 A 恰好从 1 变为 0 时，由于反相器存在延时，所以或非门输出就会瞬变为 1。因此，此电路存在静态 0 冒险。这两种冒险的出现是由于 A 和 A'的值在 A 改变后的一小段时间内相同所致。



(a)具有静态 1 冒险的简单电路　　(b)具有静态 0 冒险的简单电路

图 1.9　包含冒险的简单电路

在实现与或表达式时，如果只有一个输入变量不同的两个最小项没有被同一个蕴涵项所涵盖，那么该电路就具有静态 1 冒险。图 1.10(a)给出了另一个存在静态 1 冒险的电路。如果 $A = C = 1$，当 B 从 1 变为 0 时，输出应该恒定为 1。然而从图 1.10(b)中可以看出，如果每个门有 10 ns 的延迟，则在 D 变为 1 之前 E 会变为 0，这样就会在输出端产生一个瞬时值 0（即输出 F 具有 1 冒险）。从卡诺图中可以看出，没有同时圈定最小项 ABC 和 $AB'C$ 的一个圈。因此，如果 $A = C = 1$，且 B 从 1 变为 0，则 BC 立即变为 0，而 B 经过反相器延迟后，才使 AB' 变为 1。所以，这两个最小项的值可能瞬时均为 0，这就会导致输出 F 出现一个“毛刺”。如果我们在卡诺图中增加一个对应 AC 项的圈，并且在电路中添加相应的门[如图 1.10(c)所示]就会消除这种冒险。当 B 发生变化时 AC 保持 1 不变，这样输出就不会出现“毛刺”。一般来说，非最简式可以用来消除静冒险。



(a) 具有 1 冒险的电路　　(b) 时序图

(c) 电路冒险的消除

图 1.10　1 冒险的消除

如果想设计一个无静态和动态冒险的电路，可以使用以下步骤。

1. 找出输出的与或表达式(F')，要求每一对相邻的 1 都可以被一个蕴涵项所涵盖（所有质蕴涵项组成的与或表达式总满足此条件）。基于此表达式 F' 的两级与-或电路将没有 1 冒险、0 冒险和动态冒险。
2. 如果希望得到不同形式的电路，利用简单的因式分解或摩根定律等，将 F'转换到想要的形式。注意，将每个 x_i 和 x_i' 视为独立的变量，以防止引入冒险。

同样，也可以先找出输出的或与表达式，要求每一对相邻的 0 都可以被一个蕴涵项所涵盖，然后按照上面的方法设计一个没有冒险的两级或-与电路。

对于一个给定的电路，我们可以判别该电路是否具有静态冒险。先根据电路直接写出输出表达式（把 x_i 和 x_i' 视为独立变量）再转化为与或表达式，之后构建一个卡诺图，并把每个蕴涵项所对应的最小项圈出。如果任何一对相邻的 1 没有被一个圈覆盖，则该电路具有静态 1 冒险；同样，通过写出电路的或与表达式，我们也可以判别其是否具有静态 0 冒险。

1.6　触发器和锁存器

时序电路一般使用触发器做存储器。触发器的种类很多，比如 D 触发器、J-K 触发器、T 触发器等。时钟沿触发的 D 触发器如图 1.11 所示，此触发器在输入时钟的上升沿响应改变状态。上升沿过后，触发器的下一个状态值等于上升沿前输入 D 的值。因此触发器的特征方程为 $Q^+ = D$。其中 Q^+ 表示在时钟有效沿过后的下一个状态，D 表示时钟有效沿到来之前的输入。

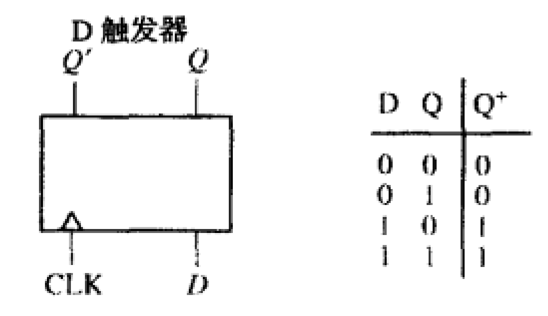


图 1.11　上升沿触发的时钟 D 触发器

图 1.12 表示时钟边触发的 J-K 触发器和它的真值表。图中，在时钟输入端处有一个圆圈，这说明该触发器是在时钟输入下降沿发生状态改变。如果 $J = K = 0$，状态不发生改变；如果 $J = 1$、$K = 0$，不论当前为何状态，触发器均置为 1；如果 $J = 0$ 且 $K = 1$，触发器总是置为 0；如果 $J = K = 1$，触发器状态翻转。由图 1.11 中真值表和卡诺图可以得到该触发器的特征方程为

$$Q^+ = J\,Q' + K'Q \tag{1.33}$$

对于时钟沿触发的 T 触发器（参见图 1.12），如果 $T = 1$，在时钟输入有效沿状态发生改变；如果 $T = 0$，状态不变。T 触发器在设计计数器时尤为有用。T 触发器的特征表达式为

$$Q^+ = QT' + Q'T = Q \oplus T \tag{1.34}$$

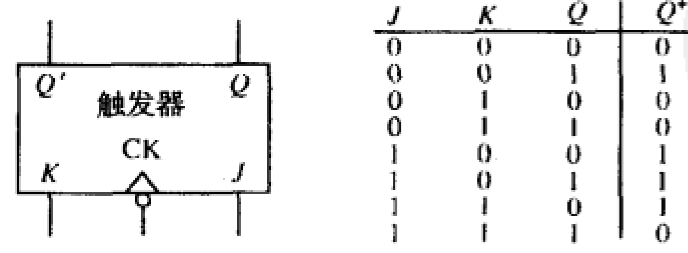


图 1.12　时钟 J-K 触发器

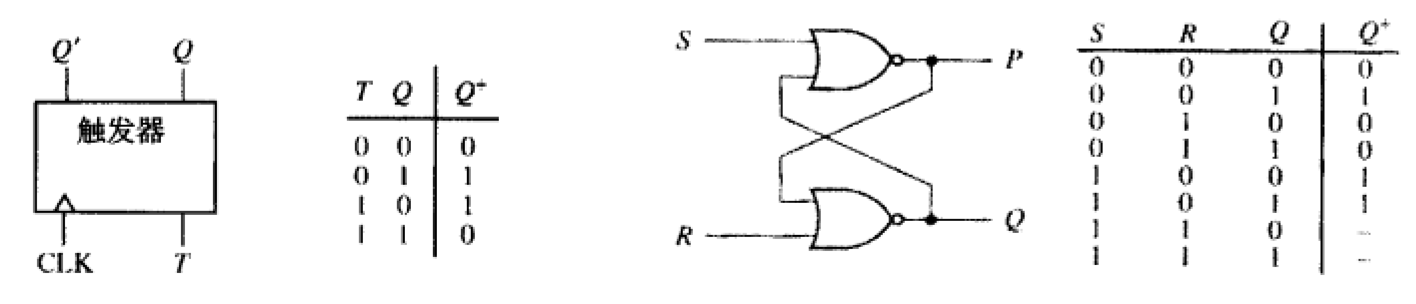
如果把 J-K 触发器中的 J 和 K 连在一起作为 T 端，则 J-K 触发器就直接转化为 T 触发器。用 T 替换式(1.33)中 J 和 K，即可得出式(1.34)。

两个或非门可以连接构成非时钟控制的 S-R（置位-复位）触发器，如图 1.14 所示。这种非

时钟控制的触发器一般称为 S-R 锁存器。如果 $S=1$ 且 $R=0$，则输出 Q 变为 1 且 $P=Q'$；如果 $S=0$ 且 $R=1$，则 Q 变成 0 且 $P=Q'$；如果 $S=R=0$，则状态不变。当 $R=S=1$ 时，$P=Q=0$，它就不能作为触发器的状态，因为触发器的两个输出应该总是互补的；如果 $R=S=1$ 且当这两个输入同时变为 0，则输出不确定（甚至振荡）。因此，S 和 R 不允许同时为 1。我们假定 $S=R=1$ 决不发生，由此得到特征方程为

$$Q^+=S+R'Q$$

此时 Q^+ 表示输入的变化引起的下一个输出状态。



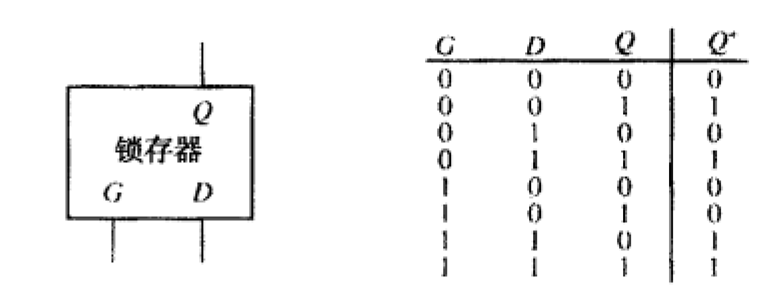
T	Q	Q^+
0	0	0
0	1	1
1	0	1
1	1	0

图 1.13　时钟 T 触发器

S	R	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	–
1	1	1	–

图 1.14　S-R 锁存器

门控 D 锁存器（如图 1.15 所示），通常也称为透明 D 锁存器。其工作状况如下：如果 $G=1$，则输出 Q 跟随输入 D ($Q^+=D$)；如果 $G=0$，则保存前一个输出值 Q ($Q^+=Q$)。除非 $G=1$，该锁存器是不会响应输入的任何变化，它只是简单地锁存 G 变为 0 之前的输入。所以有时候称 D 锁存器为电平触发 D 触发器。实际上，若把门控输入 G 看做时钟，则该锁存器在时钟的高电平响应，而在时钟的低电平不响应。该锁存器的特征方程为 $Q^+=GD+G'Q$。图 1.16 给出了基于门电路实现的 D 锁存器。因为 Q^+方程存在 1 冒险，所以通过使用额外的与门来消除该冒险。



G	D	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

图 1.15　门限控制 D 触发器

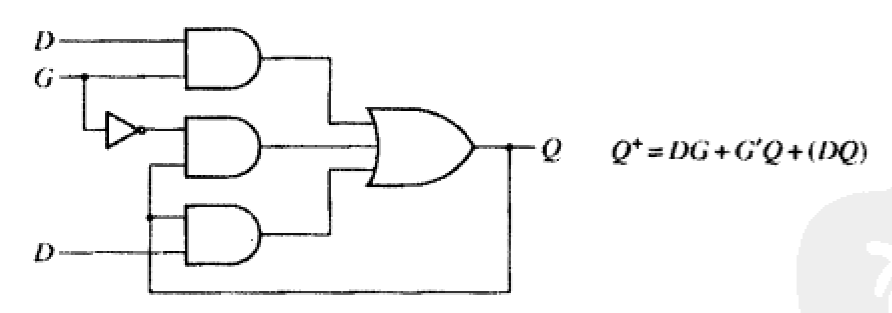


图 1.16　D 锁存器的具体实现

1.7　Mealy 时序电路设计

时序电路有两个基本类型：Mealy 电路和 Moore 电路。在 Mealy 电路中，输出同时依赖于前状态和当前输入。在 Moore 电路中，输出只取决于当前状态。Mealy 时序电路的一般组成如图 1.17 所示：一个组合逻辑电路（生成输出和下个状态）和一个状态寄存器（保存当前状态）。状态寄存器一般由 D 触发器组成。Mealy 电路的工作流程为：(1)输入 X 赋新值；(2)经过延迟后，组合电路输出 Z 和下一个状态值；(3)下一个状态值由时钟触发锁存到状态寄存器以改变当前状态。新

的状态又反馈到组合电路，重复上面的过程。

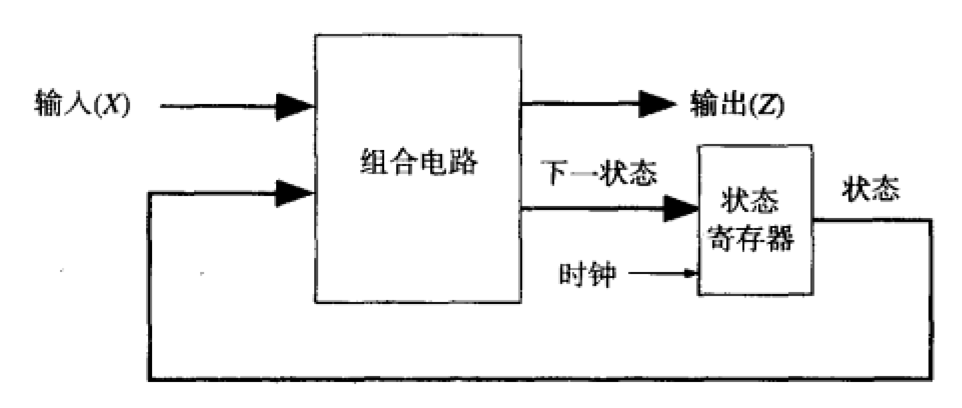


图 1.17　Mealy 时序机的一般模型

1.7.1　Mealy 时序电路设计例子 1：序列检测器

为了说明时钟沿触发的 Mealy 时序电路设计，我们先设计序列检测器，其框图如图 1.18 所示。

本电路将会检查输入的一串 0, 1 序列，记为 X，当输入 X 中含有 101 时，电路的输出 Z=1。输入 X 只能在时钟脉冲间隔期发生改变。当输入串 101 中最后一个 1 出现时，与此同步就有输出 Z=1，且 Z=1 时电路不复位。一个典型输入序列和对应的输出序列如下：

$$X=0\quad 0\quad 1\quad 1\quad 0\quad 1\quad 1\quad 0\quad 0\quad 1\quad 0\quad 1\quad 0\quad 1\quad 0\quad 0$$
$$Z=0\quad 0\quad 0\quad 0\quad 0\quad 1\quad 0\quad 0\quad 0\quad 0\quad 0\quad 1\quad 0\quad 1\quad 0\quad 0$$

我们为该序列检测器建立状态图。我们从复位状态开始，记为 S_0。如果接收到的输入为 0，那么状态不发生改变（仍为 S_0），因为我们要检测的序列模式是从 1 开始；如果接收到的输入为 1，则电路将会进入一个新状态，记为 S_1。在处于状态 S_1 时，如果接收到的输入为 0，则电路必须进入一个新状态（S_2），表明接收到了需要检测的输入序列的前两个输入（10）。在状态 S_2 下，如果接收到的输入为 1，则需要检测的输入序列已经全部接收到，输出应该为 1。此输出将作为 Mealy 电路的一个输出，并随着检测序列中最后一个 1 的出现而出现。由于我们是在设计一个 Mealy 电路，所以就不必再转换到另一个新的状态以表明接收到了检测序列 101。如果在状态 S_2 下接收到一个 1，由于电路不能在每个检测序列后都复位，所以电路不能回到起始状态。但是，序列中的最后一个 1 也可以看做是下一个序列的第一个 1，所以电路可以回到状态 S_1。说明此部分的状态图如图 1.19 所示。

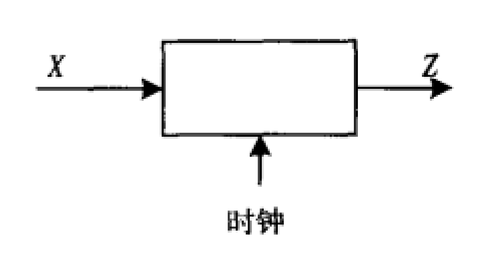


图 1.18　序列检测器框图

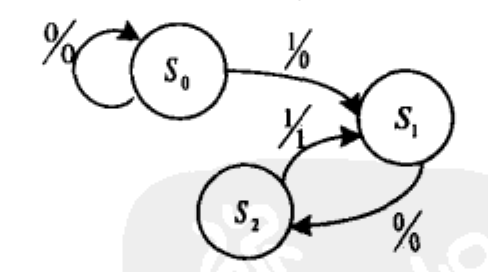


图 1.19　序列检测器的部分状态图

如果在状态 S_2 下接收到的输入为 0，这时我们已经接收了两个 0，所以电路必须复位到状态 S_0。如果在状态 S_1 下接收到的输入为 1，由于这个 1 可以作为新检测序列的第一个 1，所以状态不变仍为 S_1。完整的状态图如图 1.20 所示，状态 S_0 为起始状态，状态 S_1 表明我们接收到了一个以 1 结尾的序列，状态 S_2 表明我们接收到了一个以 10 结尾的序列。把此状态图转换为表 1.3 所示的状态表。观察表中 S_2 这一行，当输入为 1 时对应地输出为 1。

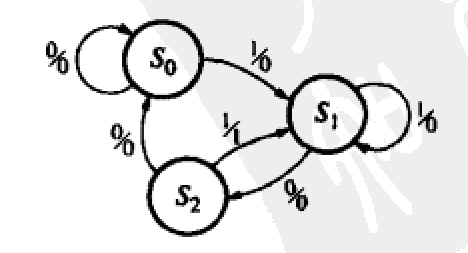


图 1.20　序列检测器的 Mealy 状态图

表 1.3　序列检测器的状态表

当前状态	下一状态		当前输出	
	$X=0$	$X=1$	$X=0$	$X=1$
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_1	0	1

下面我们进行状态赋值，即用具体的触发器的值表示具体的状态。现有的状态赋值方法有两种：（1）单热（one-hot）状态赋值；（2）编码状态赋值。在单热状态赋值中，每个状态都就使用一个触发器。因此，对于该电路来说，若使用单热状态赋值，就需要使用 3 个触发器。对于编码状态赋值来说，只需适当数目的触发器，就可以使每个状态都唯一对应这些触发器的组合。由于该电路只有 3 个状态需要赋值，所以最少需要 2 个触发器就可以表示所有的状态了。因此，在这里我们使用编码状态赋值法。设两个触发器分别为 A 和 B，令触发器状态 A=0 和 B=0 对应状态 S_0；A=0 和 B=1 对应状态 S_1；A=1 和 B=0 对应状态 S_2。因此，状态转换表见表 1.4 所示。

表 1.4　序列检测器的转移表

AB	A^+B^+		Z	
	$X=0$	$X=1$	$X=0$	$X=1$
00	00	01	0	0
01	10	01	0	0
10	00	01	0	1

从表 1.4 我们可以画出下一状态和输出 Z 的卡诺图。下一状态通常使用 A^+和 B^+ 表示，三个卡诺图如图 1.21 所示。

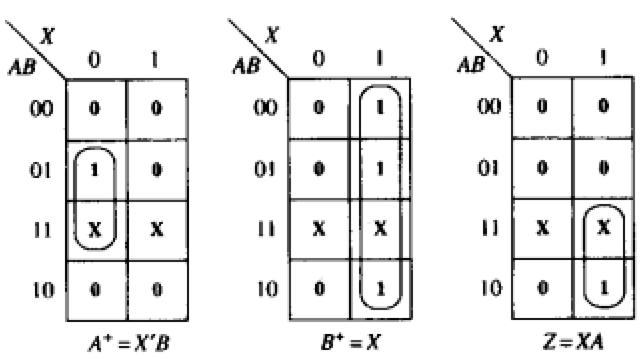


图 1.21　序列检测器下一状态和输出卡诺图

为了得到想要的下一状态，下一步我们推导触发器的输入表达式。如果使用 D 触发器，那么我们只需要把想要得到的下一时刻状态值作为当前时刻的输入值即可。因此，对于触发器 A 和 B，有 $D_A = A^+$ 和 $D_B = B^+$。序列检测器的最终电路图见图 1.22。

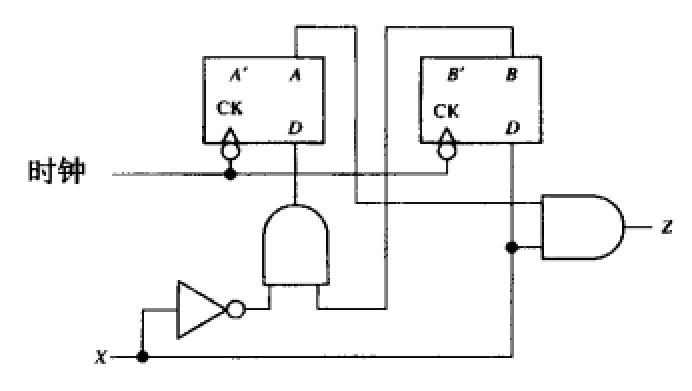


图 1.22　Mealy 序列检测器电路

1.7.2 Mealy 时序电路设计例子 2：BCD 码–余 3 码转换器

本节我们介绍一个更复杂的 Mealy 时序电路设计实例。我们将设计一个串行编码转换器，把一个 8-4-2-1 二进制编码的十进制数(BCD)转换成一个余 3 编码的十进制数。输入序列(X)和输出序列均由最低有效位开始串行输入和输出。表 1.5 列出了在 t_0, t_1, t_2 和 t_3 时刻的可能期望输入和输出。在收到四个输入之后，电路重新复位到起始状态，并准备接受另一个 BCD 码。

表 1.5　码转换器

X 输入 (BCD)				Z 输出 (余 3 码)			
t_3	t_2	t_1	t_0	t_3	t_2	t_1	t_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

把 0011 加到 BCD 码上就生成余 3 码。例如，

$$\begin{array}{r} +\ 0100 \\ \underline{0011} \\ 0111 \end{array} \qquad \begin{array}{r} +\ 0101 \\ \underline{0011} \\ 1000 \end{array}$$

如果 BCD 码的所有位同时可用，那么码转换器可以用一个四输入–四输出组合逻辑电路来实现。但是，在这里每个位是顺序到达的，即一次一位。因此我们必须用时序逻辑电路来实现此码转换器。

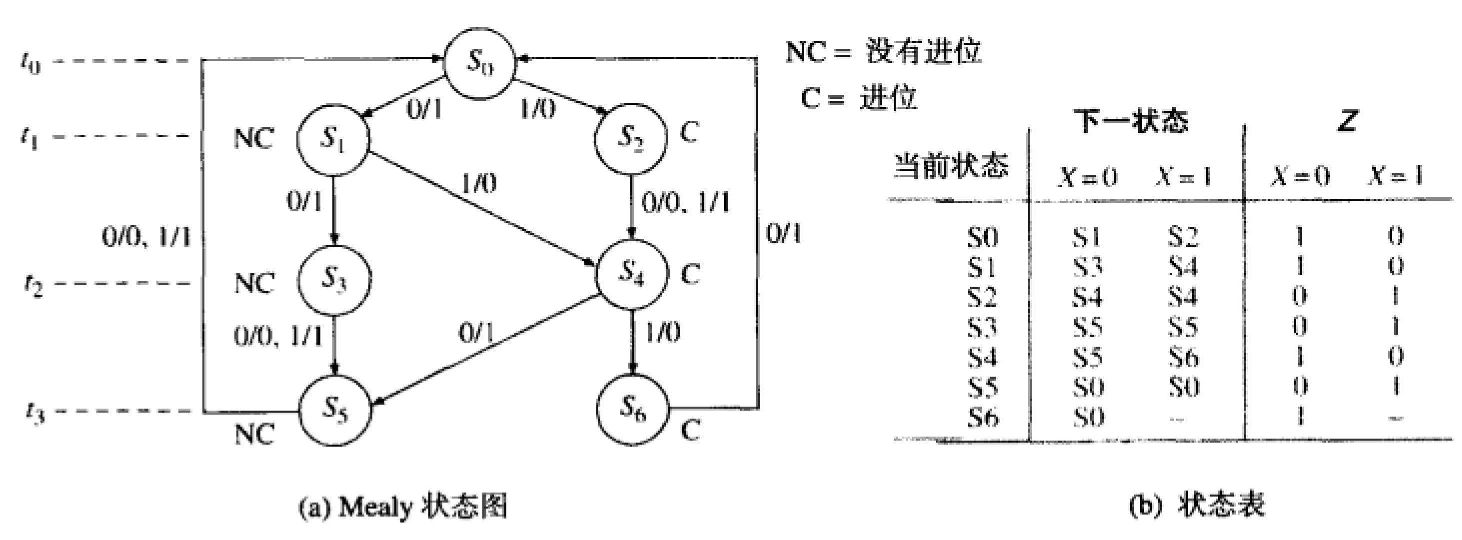
我们先建立码转换器的状态图[图 1.23(a)]。设起始状态为 S_0。由于要把 BCD 码转换成余 3 码，所以当 BCD 码的第一位到达时，我们在此位上加 1（因为 0011 的最低有效位为 1，通过加上 0011，可以把 BCD 码转换成余 3 码）。在 t_0 时刻，我们把 1 加到最低有效位，这样如果 $X = 0$，加上 1，则 $Z = 1$（没有进位）；如果 $X = 1$，加上 1，则 $Z = 0$（进位为 1）。假设状态 S_1 表示在第一次加运算之后没有进位，S_2 表明有进位。

在 t_1 时刻，我们把 1 加到下一位，因此，如果第一次加运算没有进位（状态 S_1），则若 $X = 0$，则 $Z = 0 + 1 + 0 = 1$，且没有进位（状态 S_3）；若 $X = 1$，则 $Z = 1 + 1 + 0 = 0$，且有进位（状态 S_4）。如果第一次加运算有进位（状态 S_2），那么若 $X = 0$，则 $Z = 0 + 1 + 1 = 0$，且有进位(S_4)；若 $X = 1$，则 $Z = 1 + 1 + 1 = 1$，且有进位(S_4)。

在 t_2 时刻，X 赋 0，状态转为 S_5（没有进位），状态 S_6 可以通过与上述过程相似的过程得出。在 t_3 时刻，X 再一次赋 0，电路复位成状态 S_0。

图 1.23(b)给出了与状态图相对应的状态表。此时，我们应该查证状态表中的状态数是否为最小（参见 1.9 节）。然后必须进行状态赋值，因为这一状态表共有 7 个状态，所以若使用编码

赋值，需要 3 个触发器以实现该状态表；若使用单热赋值法，则每个状态需要一个触发器，因此共需要使用 7 个触发器来实现该状态表。下一步要完成状态赋值，使状态表中的状态与触发器的状态联系起来。在序列检测器例子中，我们直接使用二进制简单赋值。现在我们要寻找一种最佳的状态赋值。最佳的状态赋值需要考虑几个因素，在许多情况下，应以减少逻辑器件的数目为目标。对于一些可编程逻辑器件来说，简单的二进制直接赋值与其他赋值一样好。对于可编程逻辑门阵列来说，状态赋值最好使用单热赋值法。近几年来，由于一个硅片含有太多的晶体管，已经不太强调最优状态赋值的重要性。



当前状态	下一状态 $X=0$	下一状态 $X=1$	Z $X=0$	Z $X=1$
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

(a) Mealy 状态图　　(b) 状态表

图 1.23　码转换器的状态图和状态表

为了减少逻辑门的数量，我们按以下规则进行状态赋值（参见 Roth, *Fundamentals of Logic Design, Fifth Edition*, Thomson Brooks/Cole, 2004）：

I. 在给定输入的情况下有相同次态（NS）的状态应给予只有一位不同的相邻赋值。（观察状态表的列）。

II. 同一个状态的次态应给予相邻赋值（观察状态表的行）。

III. 在给定输入的情况下输出相同的状态给予相邻赋值。

按以上方法赋值就趋向于在卡诺图上对应下一个状态或输出的所有 1 聚集在一起。按照以上规则，下列状态应赋给相邻值：

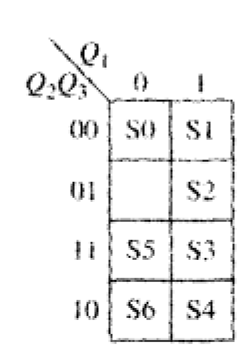
I. (1, 2)，(3, 4)，(5, 6)（在 $X=1$ 列，S_1 和 S_2 的下一个状态均为 S_4；在 $X=0$ 列，S_3 和 S_4 的下一个状态均为 S_5，S_5 和 S_6 的下一个状态均为 S_0）。

II. (1, 2)，(3, 4)，(5, 6)（S_1 和 S_2 是 S_0 的下一个状态；S_3 和 S_4 是 S_1 的下一个状态；S_5 和 S_6 是 S_4 的下一个状态）。

III. (0, 1, 4, 6)，(2, 3, 5)。

图 1.24(a)给出了满足以上规则的赋值图和对应的转换表。由于没有使用状态 001，所以对其下一状态和输出均没有要求。从图 1.25，我们可以得到下一状态和输出的表达式。图 1.26 为使用与非门和 D 触发器具体实现的码转换器。

如果用 J-K 触发器代替 D 触发器，J-K 触发器的输入方程可以由下一状态卡诺图中得到。如果给定当前状态为 Q 和下一状态为 Q^+，则 J-K 触发器的输入可以由表 1.6 确定，此表也称为激励表（此表由图 1.12 的真值表推得）。



(a) 赋值图

$Q_1Q_2Q_3$	$Q_1^+Q_2^+Q_3^+$ $X=0$	$X=1$	Z $X=0$	$X=1$
000	100	101	1	0
100	111	110	1	0
101	110	110	0	1
111	011	011	0	1
110	011	010	1	0
011	000	000	0	1
010	000	xxx	1	x
001	xxx	xxx	x	x

(b) 转换表

图 1.24 BCD 码–余 3 码转换器的状态赋值

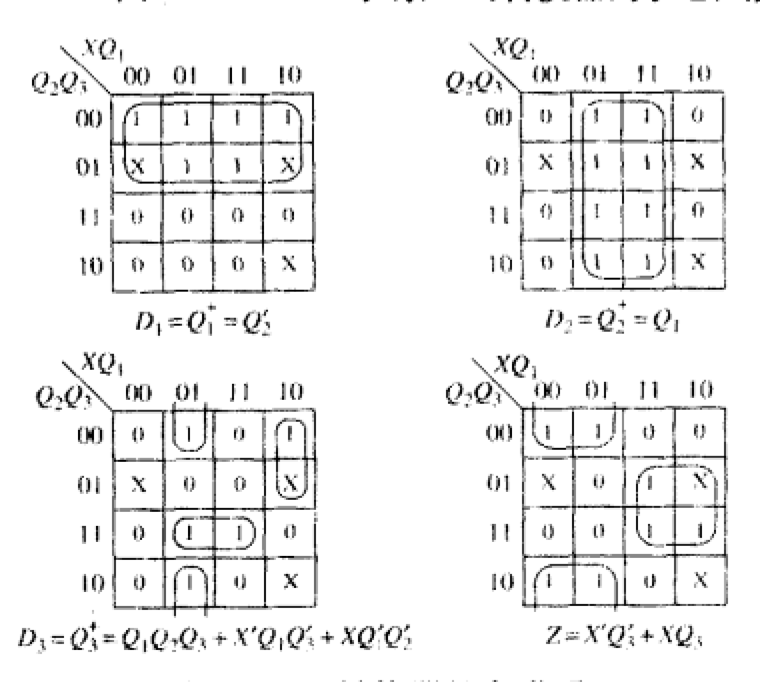


图 1.25 码转换器的卡诺图

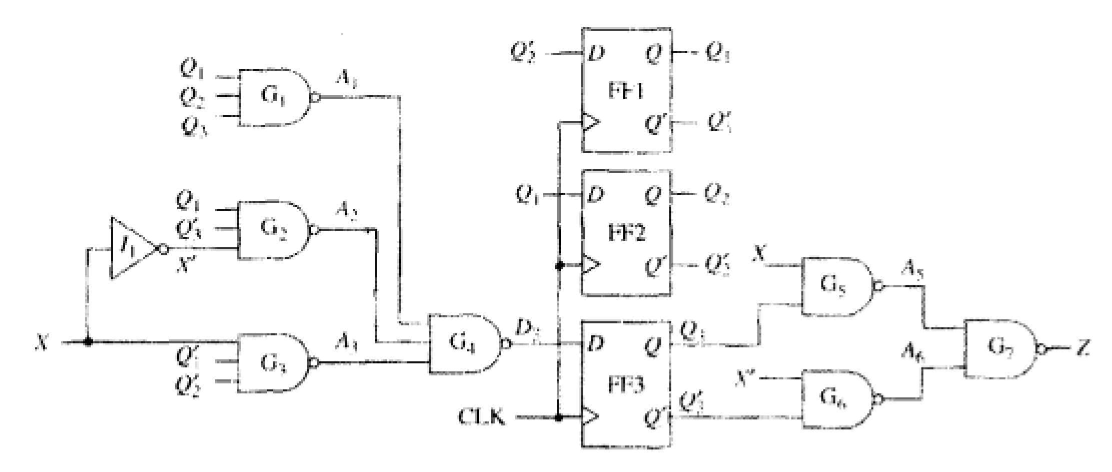


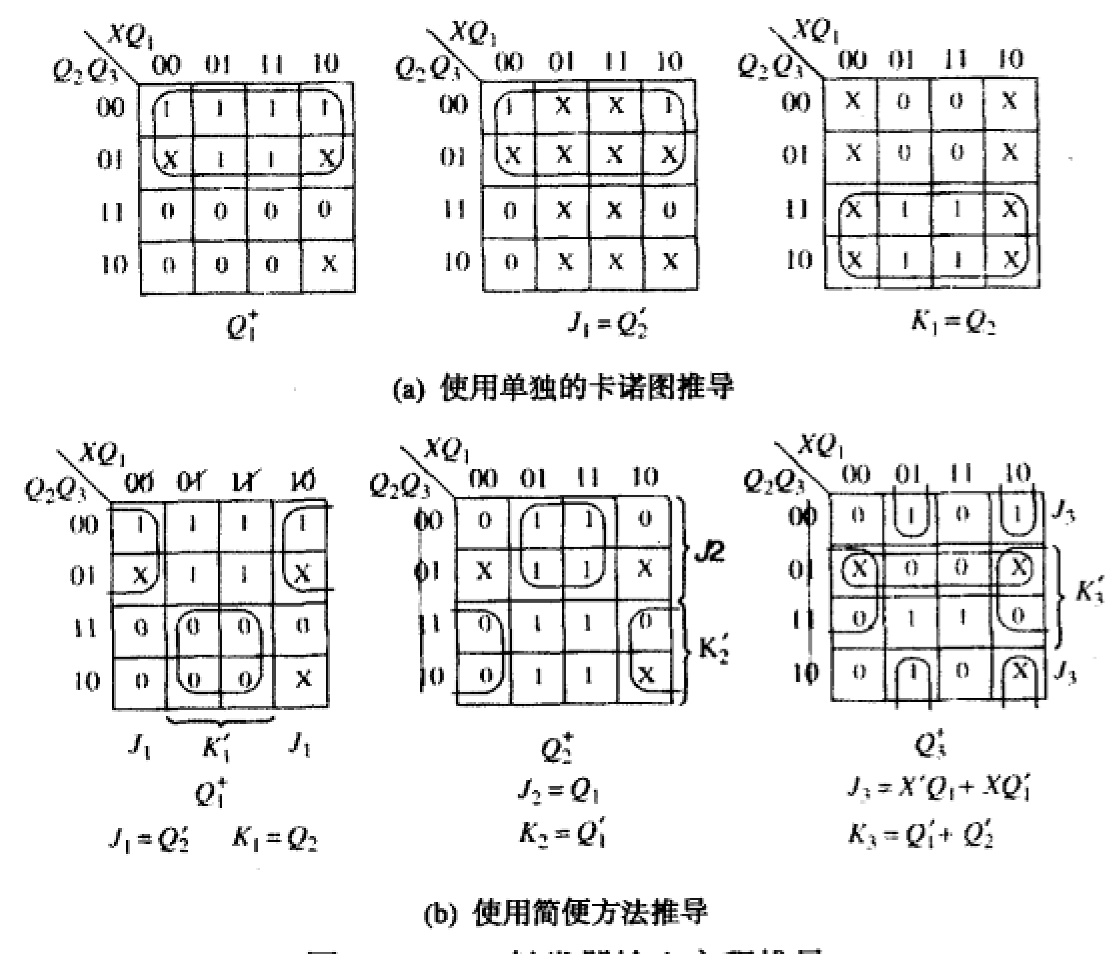
图 1.26 码转换器的具体实现

表 1.6 J-K 触发器激励表

Q	Q^+	J	K	
0	0	0	X	Q 不变，J 必为 0，K 可以为 1 时使 Q 复位（=0）
0	1	1	X	Q 变为 1，J 必为 1，使 Q 置位（=1）或反转
1	0	X	1	Q 变为 0，K 必为 1，使 Q 复位或反转
1	1	X	0	Q 不变，K 必为 0，J 可以为 1 使 Q 置位

根据图 1.23 所示状态表和图 1.24 的状态赋值，可以推得 J-K 触发器的输入方程（见图 1.27）。首先，我们利用 Q_1^+ 卡诺图得到 J-K 触发器 Q_1 的输入方程。从上一表中可知，只要 $Q_1^+=0$，则就有 $J=Q_1^+, K=X$。因此，我们在与 $Q_1=0$ 对应的一半 J_1 卡诺图中填入 Q_1^+ 卡诺图中与 $Q_1=0$ 对应的相应取值，而在与 $Q_1=0$ 对应的一半 K_1 卡诺图中则都要填入 X。当 $Q_1=1$ 时，$J_1=X$ 和 $K_1=(Q_1^+)'$。

因此，我们在与 Q_1=1 对应的一半 J_1 卡诺图中都填入 X ，而在与 Q_1=1 对应的一半 K_1 卡诺图中，则把 Q_1^+ 图中 Q_1 =1 处的数值取反后再填入。由于 J 和 K 卡诺图中都有一半是随意项，所以我们可以不必单独画出 J 和 K 卡诺图，而是直接从 Q_1^+ 卡诺图中读出 J 和 K，如图 1.27(b)所示。这一简便方法遵循以下原则：如果 Q=0，则 $J=Q^+$，因此在 Q_1^+ 卡诺图中与 Q=0 对应的一半方格中圈 1 即可得到 J。如果 Q=1，则 $K=(Q^+)'$，因此在 Q_1^+ 卡诺图中与 Q=1 对应的一半方格中圈 0 即可得到 K。 J 和 K 的表达式与 Q 无关，因为当求 J 和 K 表达式的时候。Q 被设定成了常数 0 或 1。为了在图中方便的得出 J 和 K 的表达式，我们把图中 Q 的值叉掉。实际上，这种简便方法就是把四变量的 Q^+卡诺图变为两个三变量卡诺图（一个对应 Q=0，另一个对应于 Q=1）。



(a) 使用单独的卡诺图推导

(b) 使用简便方法推导

图 1.27　J-K 触发器输入方程推导

下面介归纳了设计时序电路的一般步骤：

1. 根据给定的设计要求，确定输入和输出序列之间所需的关系，然后建立状态图和状态表。
2. 化简状态表使其状态数最小。首先，通过行匹配去掉重复的行，然后建立状态转换表，按 1.9 节的方法去做。
3. 如果化简后的状态表中有 m 个状态($2^{n-1} < m \leqslant 2^n$)，则需要 n 个触发器。使每个触发器状态组合唯一对应简化状态表中的每一个状态。这就是状态的编码赋值技术。另外，也可以用 m 个触发器的单热赋值方法。
4. 把简化的转化表中的每一个状态用已赋值的触发器状态取代，就可以形成状态转换表。这一状态转换表给出了输出和每一个触发器的下一个状态与这些触发器当前状态和输入之间的关系。
5. 画出每个触发器的下一状态卡诺图和输入卡诺图，求得出每一触发器的输入方程。再求出输出函数方程。
6. 基于可用的逻辑门电路，实现触发器输入方程和输出方程。
7. 通过计算机仿真或另外一种方法检验设计是否正确。

步骤 2 ~ 7 一般可以用 CAD 程序实现。

1.8　Moore 时序电路设计

在 Moore 时序电路中，输出只与其当前状态有关。与功能等价的 Mealy 电路相比，Moore 电路更容易设计和调试，但是它往往需要更多的状态。Moore 电路的输出值，不是在状态转移过程中产生，而是完全由状态本身决定。

1.8.1　Moore 电路例子 1：序列检测器

作为一个例子，下面我们用 Moore 电路来设计 1.7.1 节中介绍的序列检测器。该检测器对输入的 0, 1 序列 X 进行检测，当且仅当输入 X 中有 101 时，电路输出 Z 置 1。设输入 X 只能在两个时钟脉冲间隔改变取值，且当输出 $Z = 1$ 时电路不复位。

与 Mealy 电路相同，首先我们设定起始状态为 S_0（也为复位状态，如图 1.28 所示）。如果接收到的输入为 0，则状态不发生改变仍为 S_0，因此我们要寻找的序列不是从 0 开始的；但是，如果接收到输入为 1，则状态改变为一个新状态 S_1。在状态 S_1 时，如果接收到输入信号为 0，电路状态必须变为另一个新状态 S_2，以表示已接收到期望的输入序列 10。在状态 S_2 时，如果再接收到输入信号为 1，那么电路应变为另一个新状态 S_3，以表示接收到了全部要检测的输入序列，此时输出置为 1。电路处于状态为 S_0，S_1 或 S_2 时，输出均为 0。输入序列为 100 时，电路重新回到初始状态 S_0；而输入序列为 1010 时，电路最后处于状态 S_2。因为在此时如果下一个输入为 1，则输出 Z 将会再次置为 1。

图 1.28　Moore 序列检测器的状态图

表 1.7 给出了与该电路所对应的状态表。注意到由于 Moore 电路的输出只与当前状态有关，与输入 X 无关，所以该表中输出只有一列。同时我们注意到，与表 1-3 给出的完成相同功能的 Mealy 电路序列检测器相比，Moore 电路序列检测器需要多一个状态。

由于有 4 个状态，所以需要用 2 个触发器来实现该电路。如果状态赋值采用：$AB = 00$ 表示状态 S_0、$AB = 01$ 表示状态 S_1、$AB = 11$ 表示状态 S_2 和 $AB = 10$ 表示状态 S_3，则可以得到表 1.8 给出的状态转移表。

表 1.7　序列检测器的状态表

当前状态	下一状态 $X=0$	下一状态 $X=1$	当前输出 Z
S_0	S_0	S_1	0
S_1	S_2	S_1	0
S_2	S_0	S_3	0
S_3	S_2	S_1	1

表 1.8　Moore 序列检测器的转移表

AB	A^+B^+ $X=0$	A^+B^+ $X1$	Z
00	00	01	0
01	11	01	0
11	00	10	0
10	11	01	1

输出表达式为 $Z = AB'$。输出 Z 只与触发器的状态有关，与输入 X 无关。但是对应的 Mealy 电路，其输出 Z 是输入 X 的函数（如图 1.21 所示 $Z=AX$）。通过状态转换表，我们可以画出每个触发器的下一状态的卡诺图，并导出其输入方程。

1.8.2 Moore 电路设计例子 2：非归零码-曼彻斯特码转换器

作为 Moore 电路设计的另一个例子，我们将设计串行数据转换器。二进制数据常以比特流的形式在计算机之间传输。图 1.22 给出了比特流的三种不同的编码方式，设待传输的比特流为 0，1，1，1，0，0，1，0。在非归零（NRZ）编码中每比特数据占用一比特时间，没有任何改变。相反在归零（RZ）编码中，数字 0 在一个完整的比特时间内以低电平 0 表示，而数字 1 在前半个比特时间内保持高电平 1，而在后半个别特时间内则回到低电平 0。对于曼彻斯特码，传输 0 时，前半个比特时间为 0，后半个比特时间为 1；传输 1 时，前半个比特时间为 1，后半个比特时间为 0。因此，曼彻斯特码序列总是在一个比特时间的中间时刻发生改变。

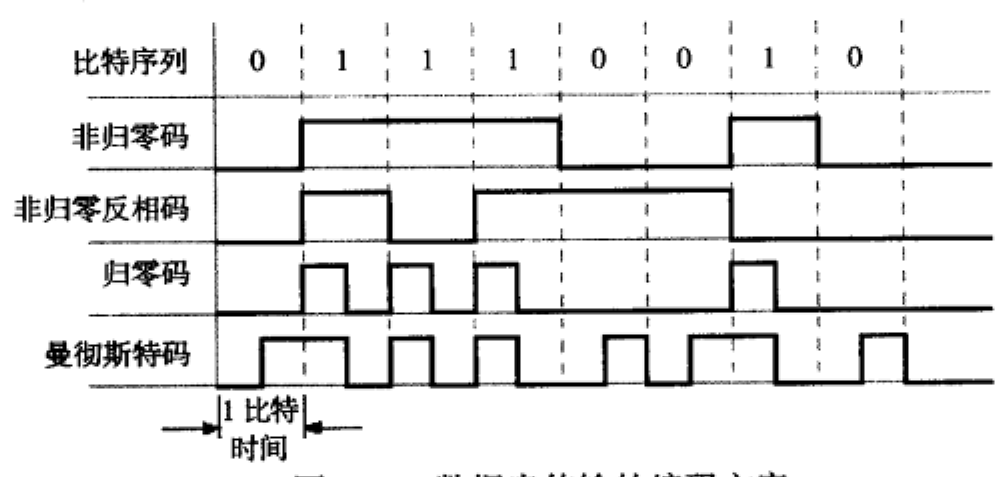
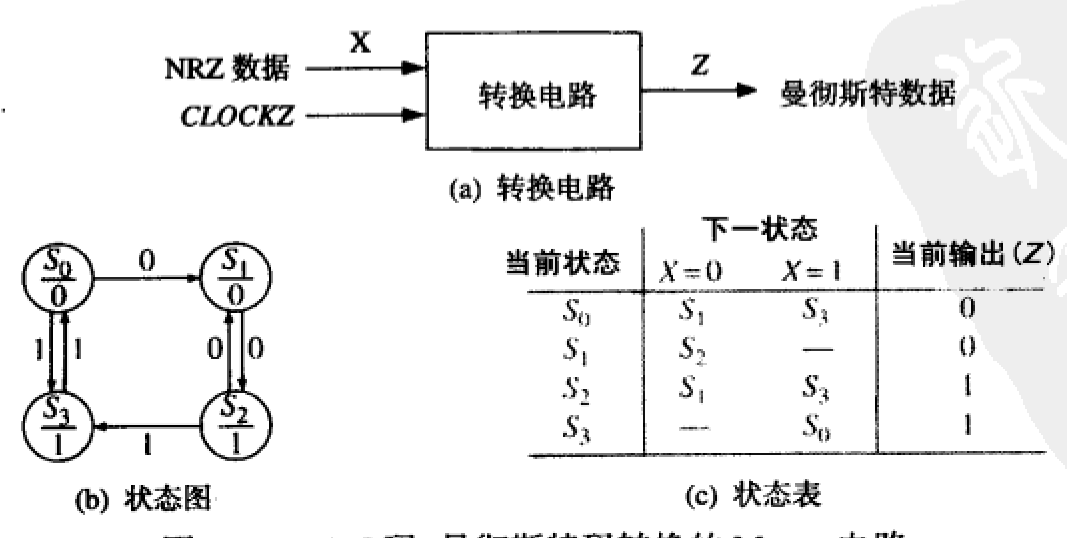


图 1.29 数据串传输的编码方案

下面我们设计一个 Moore 时序电路，把 NRZ 编码序列转换成曼彻斯特编码序列（见图 1.30）。为此，我们将使用频率为基本比特频率二倍的时钟（*CLOCK*2）。如果所传输的 NRZ 码序列是 0，那么它将持续两个 *CLOCK*2 时钟周期；如果是 1，它也将持续两个 *CLOCK*2 时钟周期。因此，在这种时钟脉冲下，从状态 S_0 开始，输入序列只能是 00 或 11，而且对应的输出序列是 01 或 10。当收到第一个 0 时，电路达到状态 S_1，并输出 0；当收到第二个 0 时，电路达到状态 S_2，并输出 1。同样，从状态 S_0 开始，如果收到第一个 1，电路达到状态 S_3，并输出 1；当收到第二个 1 的时候，电路必须转移到输出为 0 的状态，此状态 S_0 比较合适，因为其输出为 0，且电路又处于准备接收下一个 00 或 11 序列。如果在状态 S_2 下收到 00 序列，电路可以转移到状态 S_1 然后返回到状态 S_0；如果在状态 S_2 下收到 11 序列，电路可以转移到状态 S_3 然后返到状态 S_0。与状态图相对应的状态表有两个随意项，对应于两个不可能发生的输入序列。



当前状态	下一状态 $X=0$	$X=1$	当前输出(Z)
S_0	S_1	S_3	0
S_1	S_2	—	0
S_2	S_1	S_3	1
S_3	—	S_0	1

(c) 状态表

图 1.30 NRZ 码–曼彻斯特码转换的 Moore 电路

图 1.31 为 Moore 电路的时序图。我们注意到，对应于每个 NRZ 码输入，输出曼彻斯特码延时一个时钟周期。这种延时是由于 Moore 电路只能在时钟脉冲有效沿到来时才对输入响应的缘故。这与 Mealy 电路不同，Mealy 电路的输出能在下一个时钟到来之前随着输入的变化而变化。

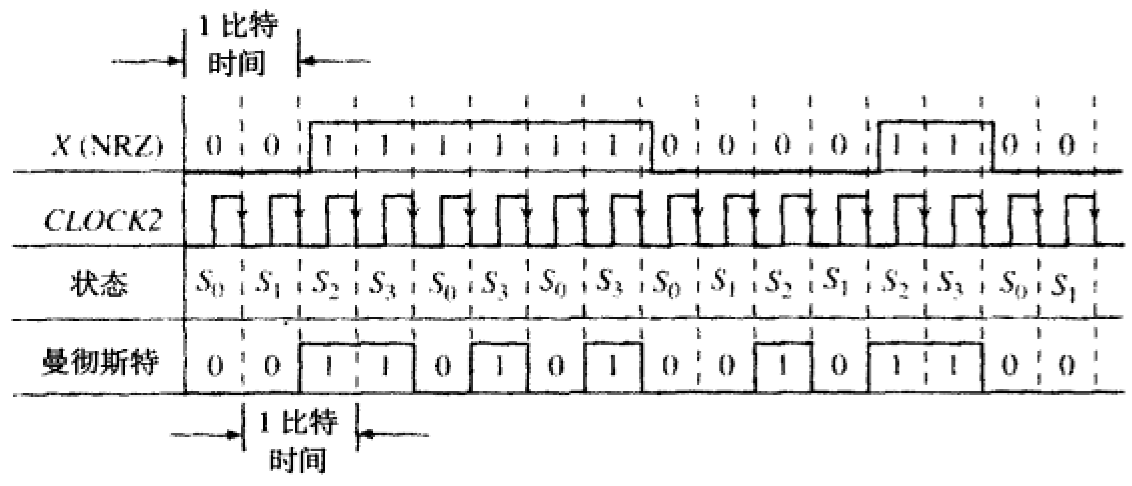


图 1.31 Moore 电路时序图

1.9 等价状态和状态表化简

等价状态的概念对于设计和测试时序电路十分重要，并有益于降低电路硬件成本。如果我们通过观察输入和输出序列不能区分时序电路中的两个状态，则这两个状态称为等价状态。下面以两个时序电路 N_1 和 N_2（见图 1.32）为例，N_1 和 N_2 可以是同一个电路的两个副本，N_1 从状态 s_i 开始工作，N_2 从状态 s_j 开始工作。我们对两电路输入同一个序列 $\underline{X}$，并观察输出序列 $\underline{Z}_1$ 和 $\underline{Z}_2$（下划线表示一个序列）。如果 $\underline{Z}_1$ 和 $\underline{Z}_2$ 相同，我们把电路复位到状态 s_i 和 s_j，并输入另一不同的序列，并观察输出序列 $\underline{Z}_1$ 和 $\underline{Z}_2$。如果两个输出序列对于所有可能的输入序列都是相同的，我们就说 s_i 和 s_j 等价（$s_i \equiv s_j$）。我们可以这样严格定义等价状态：当且仅当任意输入序列 $\underline{X}$ 对应的输出序列 $\underline{Z}_1 = \lambda_1(s_i, \underline{X})$ 和 $\underline{Z}_2 = \lambda_2$（$s_j$，$\underline{X}$）都相同时，$s_i \equiv s_j$。这不是一个很实用的检查等价状态的方法，至少在理论上它需要无限长输入序列。实际上，如果我们对状态数有限制，则可以限制测试序列的长度。

判断状态等价的一个更实用的方法是使用等价状态定理：如果对于任意输入 X 输出均相同，并且下一个状态也相同，那么 $s_i \equiv s_j$。当使用等价状态的定义时，我们必须考虑全部的输入序列，但是不必考虑系统内部状态的任何信息。当使用等价状态定理时，我们必须考虑到输出和下一个状态，但是我们只需要考虑单个输入信号而不是一串输入序列。

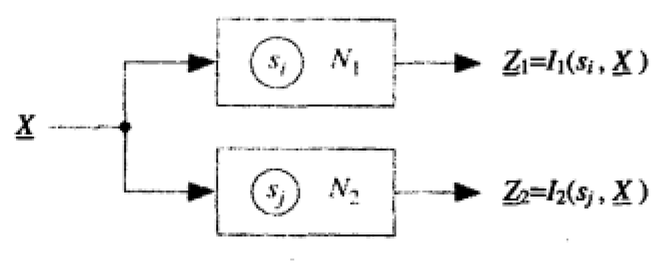


图 1.32 时序电路

图 1.33(a)的状态表可以通过消除等价状态来化简。首先，我们观察到当 $X = 0$ 和 $X = 1$ 时，状态 a 和 h 都有相同的下一个状态和输出。因此，$a \equiv h$，这样我们可以消除 h 行，并用表格中的 a 替换 h。为了判断其他状态是否也等价，我们将使用等价状态定理。从表格中可以看出，由于状态 a 和 b 的输出相同，所以当且仅当 $c \equiv d$ 且 $e \equiv f$ 时 $a \equiv b$，这时我们说 c-d 和 e-f 是 a-b 的蕴涵对(implied pairs)。为记录蕴涵对，我们制作了一个蕴涵表，如图 1.26(b)所示。我们把 c-d 和 e-f 放在行 a 和列 b 相交的方格里，表示蕴涵关系。由于状态 d 和 e 有不同的输出，我们放置一个×在 d-e 方格处以指示 $d \not\equiv e$。以这种方式完成蕴涵表后，我们再来过一遍此表。e-g 方格包含了 c-e 和 b-g；由于 c-e 方格中有一个×，所以 $c \not\equiv e$，由此可得 $e \not\equiv g$，所以我们给 e-g 方格也画×。同样，由于 $e \not\equiv f$，我们给 f-g 方格画×。我们重新过一遍该表：把所有包含蕴涵对 e-f 和 f-g 的方格画×（在图中用点线画出）。这

时，表中没有再可画×的方格，不再过一遍。由于所有包含非等价状态方格都被画×，所以剩下的方格就给出了等价状态对。从第一列得到 $a \equiv b$；从第三列得到 $c \equiv d$；从第五列得到 $e \equiv f$。

这种基于蕴涵表判定等价状态的步骤总结如下：

1. 构建一个包含所有状态对的方格表，每个方格代表一对状态。
2. 比较状态表每一行中的状态对，如果状态 i 和 j 的输出不同，放置一个×在方格 i-j 以示 $i \neq j$；如果输出相同，放置蕴涵状态对在方格 i-j 中（如果对于某输入 x，i 和 j 的下一个状态是 m 和 n，那么 m-n 是一对蕴涵状态）。如果输出和下一个状态均相同（或者 i-j 的蕴涵对是自身），则放置一个（√）在方格 i-j 中表示 $i \equiv j$。
3. 检查蕴含表的每一个方格。如果方格 i-j 包含蕴涵状态对 m-n，而方格 m-n 包含一个×，那么 $i \neq j$，并且方格 i-j 中也应该放置×。
4. 如果在第 3 步有新的×加入，重复第 3 步直到没有新的×加入。
5. 对于每一个不包含×的 i-j 方格，有 $i \equiv j$。

如果需要，可以在构建蕴涵表之前用行匹配来化简状态表。尽管我们以 Mealy 表为例，此过程对 Moore 表也同样适用。

如果第一个电路的每一个状态在第二个电路中都有一个等价状态，则称这两个时序电路是等价的，反之亦然。

与此类似的优化技术在 CAD 的工具中均得到了应用。近几年来，由于一个芯片上的晶体管数目激增，虽然状态化简显得并不太重要，但是从减少电路面积和降低功耗方面来说，显然仍起重要的作用。

当前状态	下一状态 X=0	1	当前输出 X=0	1
a	c	f	0	0
b	d	e	0	0
c	~~h~~ a	g	0	0
d	b	g	0	0
e	e	b	0	1
f	f	a	0	1
g	c	g	0	1
~~h~~	~~e~~	~~f~~	~~0~~	~~0~~

(a) 通过匹配化简状态表

	a	b	c	d	e	f
b	c-d e-f					
c	f-g	a-d e-g				
d	b-c f-g	e-g	a-b			
e	×	×	×	×		
f	×	×	×	×	a-b	
g	×	×	×	×	c-e b-g	c-f a-g

a≡b iff c≡d and e≡f

(b) 蕴涵表（第一轮）

	a	b	c	d	e	f
b	c-d e-f					
c	× b-g	× a-d e-g				
d	× b-c f-g	× e-g	a-b			
e	×	×	×	×		
f	×	×	×	×	a-b	
g	×	×	×	×	× c-e b-g	× c-f a-g

(c) 在第二轮和第三轮后

图 1.33 状态表化简

	X = 0	1	X = 0	1
a	c	e	0	0
c	a	g	0	0
e	e	a	0	1
g	c	g	0	1

(d) 最终化简表

图 1.33（续） 状态表化简

1.10 时序电路的时序

时序电路功能的正确实现涉及到几个时序问题。比如触发器、门电路和传输线的传输延迟、触发器的建立时间和保持时间、时钟同步和时钟偏移(clock skew)等。这些问题对时序电路的设计非常重要。本节中我们对各种时序电路的时序问题加以分析。

1.10.1 传输延迟、建立时间和保持时间

时钟沿触发的时序电路从输入时钟发生改变到输出 Q 发生改变需要很小一段时间，这一小段时间称为传输延迟（如图 1.34 所示）。这种传输延迟可以与输出电平的变化有关，比如从高电平变为低电平，或从低电平变为高电平。输出 Q 从低电平变为高电平时的传输延迟记为 t_{plh}，从高电平变为低电平时的传输延迟称为 t_{phl}。

对于一个理想的 D 触发器，如果输入 D 正好在时钟上升沿发生改变，触发器就会正确工作。然而对实际触发器来说，为了正常工作，其输入 D 必须在时钟上升沿到来之前的一段时间内保持稳定，这一段时间称为建立时间 t_{su}。还有 D 必须在时钟上升沿之后的一段时间内也要保持稳定，这段时间称为保持时间 t_h。图 1.34 示意了一个在时钟上升沿改变状态的 D 触发器的建立和保持时间。输入 D 可以在图 1.34 中阴影区域内的任意时刻发生改变，但是它必须在时钟沿有效之前 t_{su} 时间段和时钟沿有效之后的 t_h 时间段内保持稳定。如果 D 在这段禁止变化的时间段内发生改变，则很难确定触发器的状态是否改变，甚至更糟糕，触发器会错误运行并且输出一个短脉冲或进入震荡状态。t_{su} 和 t_h 的最小值与 t_{plh} 和 t_{phl} 的最大值都可以从厂家的数据手册中得到。

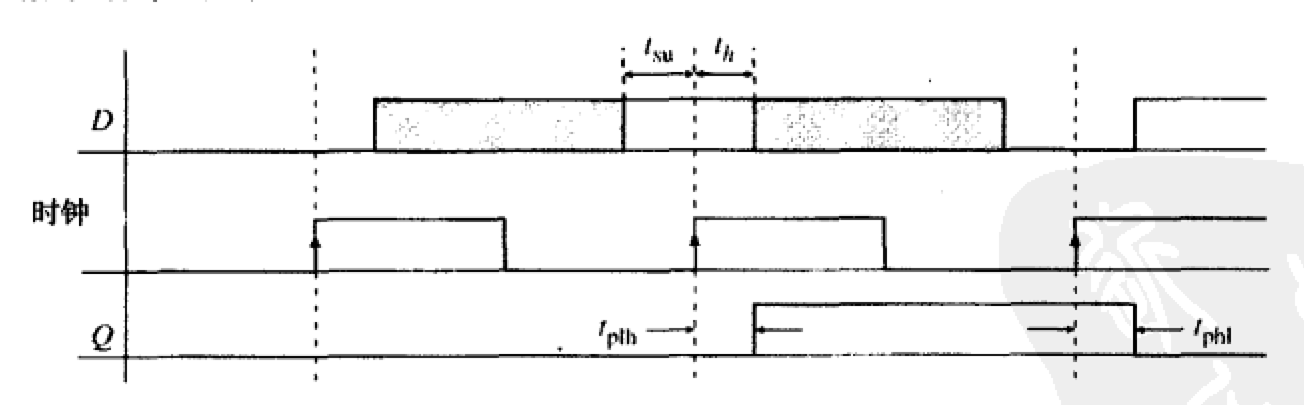


图 1.34 D 触发器启动时间和持续时间

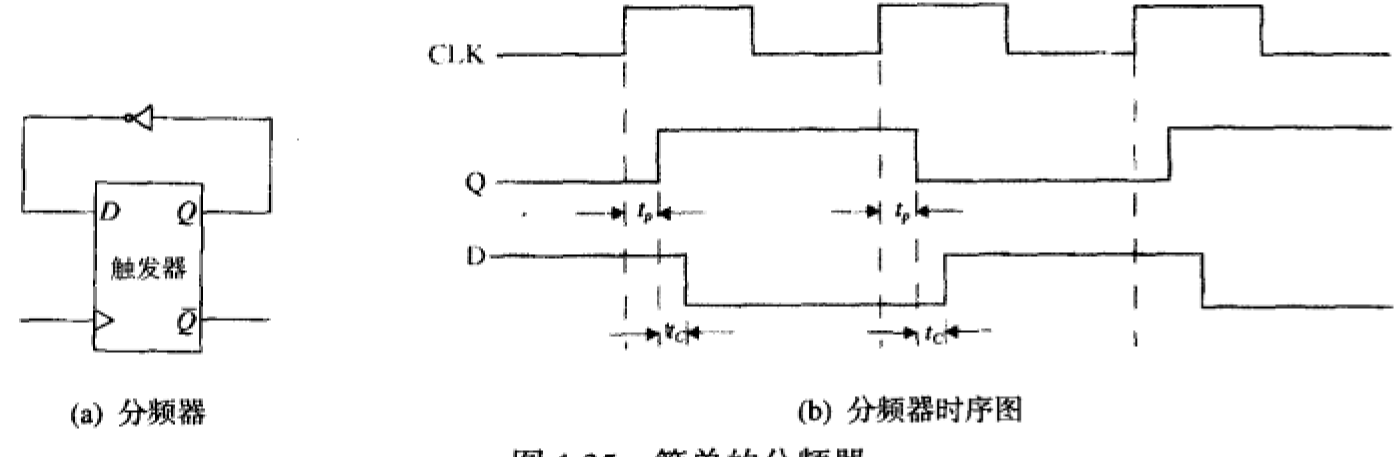
1.10.2 最大时钟工作频率

在在同步时序电路中，状态在时钟有效沿到来后马上发生改变。时序电路的最大时钟工作频率与诸多方面因素有关。首先时钟周期一定要足够长，这样才能使所有的触发器和寄存器在时钟有效沿到来之前有时间保持稳定。在时序电路中，传输延迟、建立和保持时间带来很复杂的时序

关系。

下面讨论一个简单的电路，如图 1.35(a)所示。D 触发器的输出通过一个反相器反馈到输入端。假设时钟 CLK 的波形如图 1.35(b)所示。若当前触发器的当前输出为 1，则经过一个反相器传输延迟后其输入为 0。此时若建立时间过后，时钟有效沿才到达，则触发器的输出变为 0。以此反复持续下去，触发器将输出周期为时钟周期 2 倍的时钟波形，实际上这一电路就起了一个分频器的作用。

如果我们增加一点输入时钟信号的频率，则该电路作为分频器仍工作。但是，如果我们增加输入时钟频率过大，则由于反相器的传输延迟，触发器的输入没有时间保持稳定以满足建立时间要求。同样，如果反相器足够快将取反的输出反馈回到触发器的输入端，则电路也会出现时序问题，即不能满足触发器保持时间的要求。由此可见，由于传输延迟、建立和保持时间问题，带来时序电路的各种时序问题。



(a) 分频器　　(b) 分频器时序图

图 1.35　简单的分频器

1.10.3 时序条件

对于如图 1.17 所示的一般时序电路，假设组合逻辑电路的最大传输延迟是 t_{cmax}，从时钟改变到触发器输出改变的最大传输延迟是 t_{pmax}（取 t_{plh} 和 t_{phl} 的最大值）。为了使电路可以正常工作，则必须满足以下四个要求：

1. **时钟周期要足够长，以满足触发器建立时间的要求。**时钟周期要足够长，要给触发器输出的改变和组合电路输出的改变预留时间，同时仍有时间满足建立时间的要求。有效时钟沿到来之后，经过最大值为 t_{pmax} 的传输延迟后，触发器输出才有变化；然后经过最大值为 t_{cmax} 的传输延迟后，组合电路的输出才发生变化。因此，从有效时钟沿到来之时，到输出 Q 的变化反馈到 D 触发器输入端，所用最大传输延迟为 $t_{pmax} + t_{cmax}$。为了保证触发器的正常工作，组合电路的输出还必须在时钟周期结束前一段时间 t_{su} 内保持稳定。设时钟周期为 t_{ck}，则有

$$t_{ck} \geqslant t_{pmax} + t_{cmax} + t_{su}$$

   t_{ck} 和$(t_{pmax} + t_{cmax} + t_{su})$之间的差值称为建立时间余量。

2. **时钟周期要足够长，以满足触发器保持时间的要求。**在有效时钟沿到来之际，如果输出 Q 的改变值通过组合逻辑电路很快反馈到触发器的输入端，则就会破坏保持时间的要求。保持时间应满足

$$t_{pmin} + t_{cmin} \geqslant t_h$$

在判断电路的保持时间条件时，考虑到最坏的情况，必须使用传输延迟的最小值。对于正常触发器，一般有 $t_{pmin}>t_h$ 时，因此 Q 的改变不会破坏保持时间的要求。

3. **电路外部输入的改变应满足触发器建立时间的要求。**当电路的输入 X 发生改变的时刻距有效时钟沿太近时，则破坏触发器建立时间要求。当一个时序电路的输入 X 发生改变时，我们必须确保在有效时钟沿到来之前已经传到触发器的输入端，以保证建立时间的要求。如果输入 X 在有效时钟沿到来 t_x 之前发生变化（见图 1.36），则允许通过组合逻辑电路的传输延迟要最大，并且还应有 t_{su} 余量以满足建立时间条件。所以，此时建立时间应满足

$$t_x \geqslant t_{cxmax} + t_{su}$$

其中，t_{cxmax} 是从 X 变换到触发器输入端的最大传输延迟。

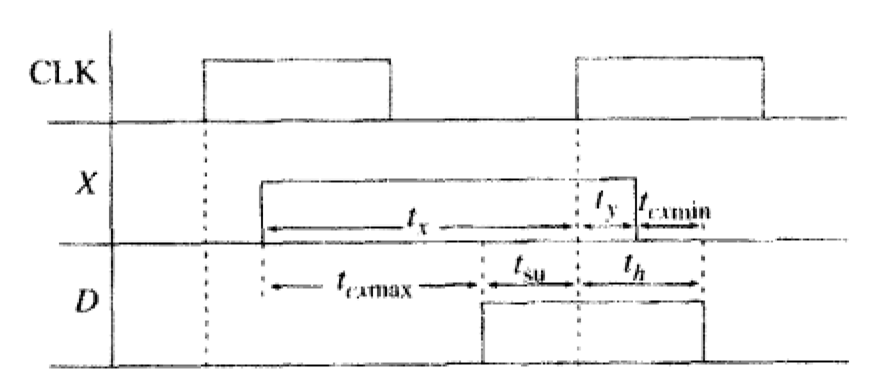


图 1.36　X 改变时的启动时间和持续时间

4. **电路外部输入的改变应满足触发器的保持时间。**为了保证保持时间，我们必须确保 X 不会在有效时钟沿过后很快发生改变。如果 X 的改变即刻传播到触发器的输入端（延迟为 0），则在有效时钟沿到来后 X 在 t_h 时间内不应发生改变，好在 X 的变化传到触发器输入端的传输延迟不会为 0。设 t_{cxmin} 是从 X 变化到触发器输入端的最小传输延迟，则 X 的变化至少在有效时钟沿到来后 t_{cxmin} 时间内，不会传到触发器的输入端。因此，如果 X 在有效时钟沿之后的 t_y 时刻发生改变，则满足保持时间的要求为

$$t_y \geqslant t_h - t_{cxmin}$$

如果 t_y 是负的，X 可以在有效时钟沿之前发生改变，并且仍能满足保持时间要求。

只要给出一个电路，我们就可以使用上面的准则确定其可靠的工作频率和输入可变的安全区域。例如，如图 1.35(a)所示的分频电路。如果反相器的最小和最大传输延迟分别为 1 ns 和 3 ns，并且 t_{pmin} 和 t_{pmax} 分别为 5 ns 和 8 ns，则我们可以通过准则（1）求出其时钟的最大工作频率。假设触发器的建立时间和保持时间分别为 4 ns 和 2 ns，则为了正确工作必须使得 $t_{ck} \geqslant t_{pmax} + t_{cmax} + t_{su}$。在本例中，触发器的 t_{pmax} 为 8 ns，t_{cmax} 为 3 ns，t_{su} 为 4 ns，故有

$$t_{ck} \geqslant 8 + 3 + 4 = 15\text{ ns}$$

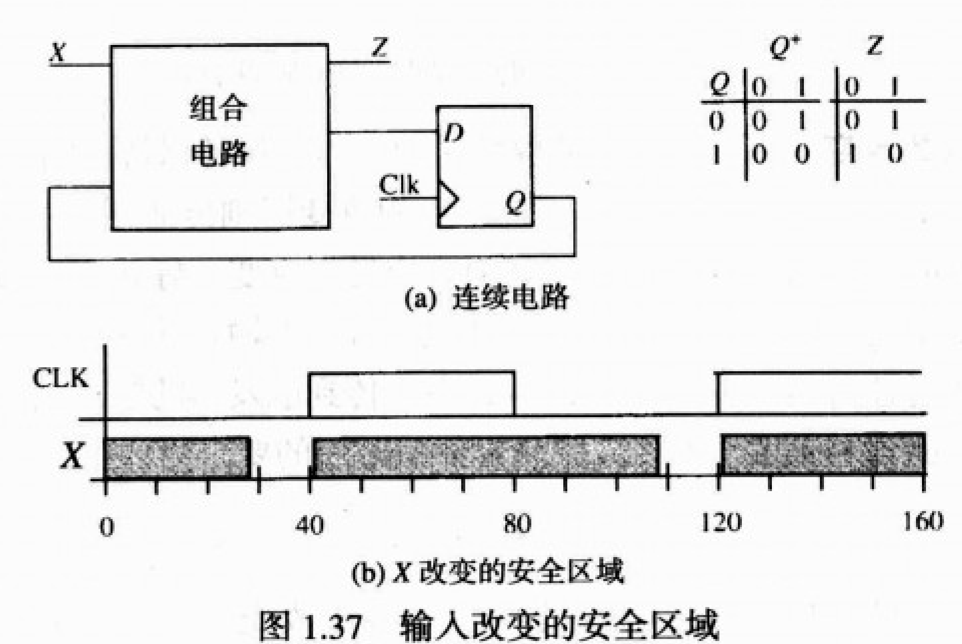
最大时钟频率为 $1/t_{ck} = 66.67$ MHz。我们还应该确定是否满足保持时间的要求。保持时间要求在有效时钟沿到来后的 2 ns 内 D 触发器的输入不能发生改变。如果 $t_{p\min} + t_{c\min} \geqslant 2\text{ns}$，则保持时间就可以得到满足。在本电路中，$t_{pmin}$ 和 t_{cmin} 分别为 5 ns 和 1 ns，这样保证了输出 Q 在有效时钟沿到来后 5 ns 内不发生改变，并且再需要至少 1 ns 的时间防止输出的变化通过反相器反馈到输入端。因此，输入 D 在有效时钟沿到来后 6 ns 内不发生改变，这样就自然可以满足保持时间要求。

由于没有外部输入，这就是唯一我们需要满足的时序约束关系。

现在，请看图 1.37(a)。假设组合电路的延迟在 2 ~ 4 ns 之间，触发器的传输延迟在 5 ~ 10 ns 之间，建立时间为 8 ns，保持时间为 3 ns。为了满足建立时间要求，时钟周期必须大于 $t_{p\max}+t_{c\max}+t_{su}$。所以

$$t_{ck} \geqslant 10+4+8=22\ \text{ns}$$

如果输出在有效时钟沿到来后 3 ns 内不发生改变，则可以满足保持时间要求。这里，输出在 $t_{p\min}+t_{c\min}$ 时间内不发生改变。由于 $t_{p\min}$ 为 5 ns，$t_{c\min}$ 为 2 ns，所以为了满足保持时间要求，输出在有效时钟沿到来后 7 ns 内不应发生改变。此电路有外部输入，所以我们要使用准则 3 和 4 为输入 X 的改变设定安全区域。输入 X 应该在有效时钟沿到来之前 $t_{cx\min}+t_{su}$（如 4 ns + 8 ns）时间内保持稳定。同样，也应该在 $t_h-t_{cx\min}$（如 3 ~ 2 ns）时间内保持稳定。这样，输入 X 应该在时钟沿到来前的 12 ns 内不发生改变，并且在有效时钟沿到来后 1 ns 内也不应发生改变。虽然保持时间为 3 ns，但是我们发现输入 X 只要在有效时钟沿到来后 1 ns 内不发生改变就可以了，这是因为输入变化至少还需要 2 ns 的延迟才能传输到触发器 D 的输入端（其中 2 ns 为组合电路的最小延迟）。图中暗影区域的波形表示输入 X 的安全改变区域，在这一区域内当 X 发生改变时，电路的工作将不会出现问题。



(a) 连续电路

(b) X 改变的安全区域

图 1.37　输入改变的安全区域

1.10.4　时序电路中的毛刺

时序电路的外部输入常常是异步的。输入的改变可以引起输出或下一状态的瞬时错误值，我们称其为下一状态毛刺或输出毛刺。比如，如果图 1.23(b)的状态表以图 1.17 的形式实现，则其时序图如图 1.38 所示。触发器的传输延迟被忽略，所以触发器的状态变化和有效时钟沿是同步的。在这一例子中，输入序列 X 为 00101001，并且 X 在时钟脉冲中间位置变化。对任意给定的时间，下一状态和输出 Z 可以从下一状态表中读出。例如，在 t_a 时刻，State = S_5, $X=0$，所以 Next State= S_0 并且 Z = 0。在时钟上升沿的 t_b 时刻，State= S_0 且 X 仍为 0，所以 Next State= S_1 且 Z = 1。然后 X 变为 1，在 t_c 时刻，Next State= S_2, Z = 0。我们可以看到在时刻 t_b 有一个窄脉冲（毛刺）（有时称为一个错误输出）。因为 X 的改变没有与有效时钟沿精确同步，所以输出 Z 在 t_b 时刻有一个错误瞬时值。如图 1.38 所示，正确的输出序列为 11100011。虽然有若干个毛刺出现在正确输出之间，但是如果 Z 在正确时刻被读出，则这些毛刺是无济于事的。在 t_b 时刻出现的下一状态的毛刺（S_1）也不会引起问题，因为在有效时钟沿时刻下一个状态的取值是正确的。

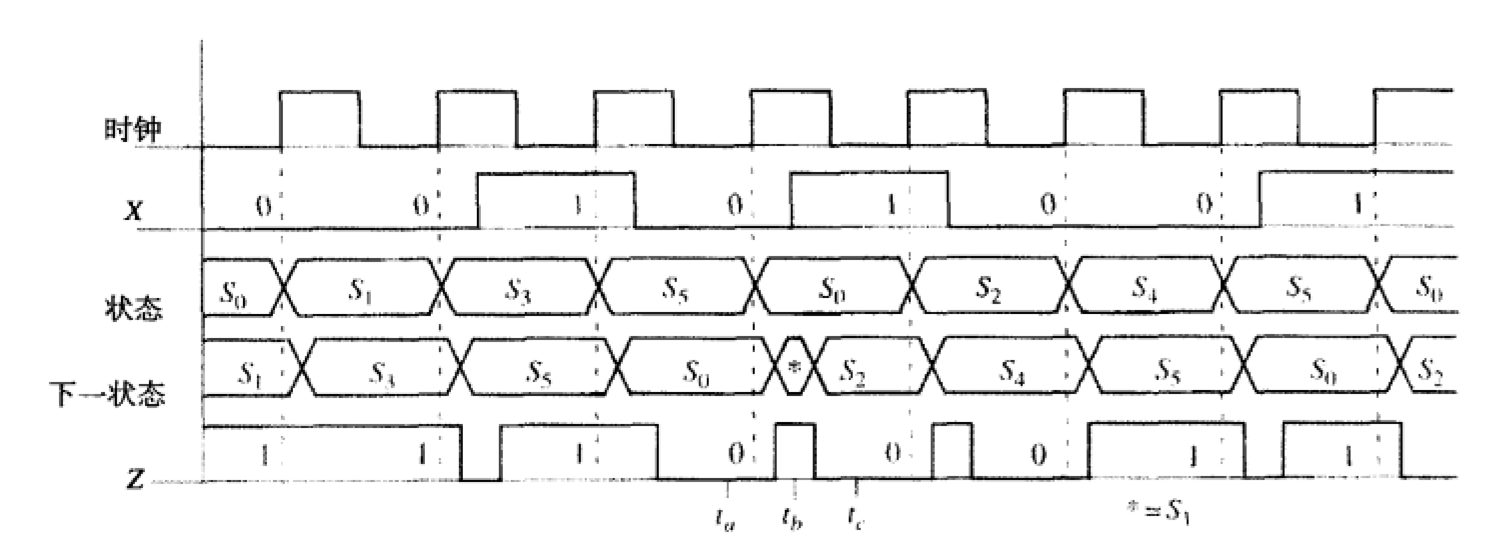


图 1.38 码转换器的时序图

图 1.26 中电路的时序图如图 1.39 所示。该图与图 1.38 的时序图类似，只是状态由三个触发器的状态所代替，并且每个门和触发器都有 10 ns 的传输延迟。

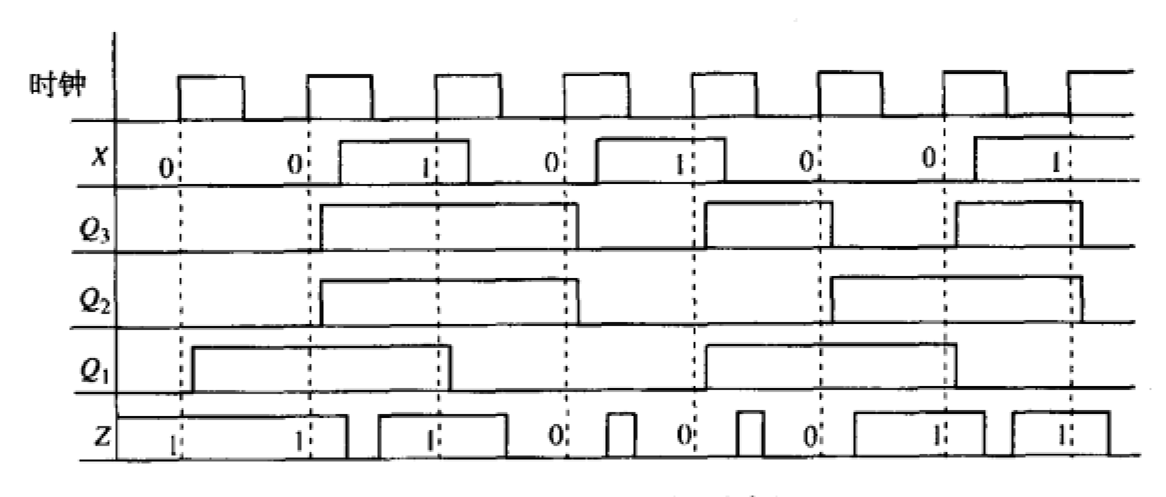


图 1.39 图 1.26 的时序图

1.10.5 同步设计

同步设计是最普遍使用的数字设计技术中的一种。它通过一个时钟对系统中所有的触发器、寄存器和计数器进行同步。同步电路要比异步电路更可靠。在同步电路中，所有的变化都跟随有效时钟沿而变化。在电路中的一个模块的输出要经过一个完整的时钟周期才能传输到电路的另一模块的输入。与异步技术相比，同步设计使设计过程和调试过程更加简单。

图 1.40 示意了一个同步数字系统。假设该系统是由多个模块或部件构成的。这些部件可以是触发器、寄存器、计数器、加法器和多路选择器等。在同步系统中，所有时序部件都是在同一个时钟下同步的。典型的数字系统可以分为控制部分和数据处理部分，图 1.40 给出的许多部件均属于数据处理部分。控制部分是一个时序电路，用于产生控制信号来控制数据处理器的工作。比如，如果数据处理器包含一个移位寄存器，控制部分则产生控制信号 *Ld*（装入）和 *Sh*（移位），决定寄存器何时被载入或者何时进行移位操作。通常，我们使用共同的时钟对控制和数据部分进行同步。数据处理部分可以产生状态信号（图 1.40 中未标示出）响应控制器。例如，如果数据处理器产生算术溢出，则数据处理部分会发出状态信号 *V* 以示溢出。控制部分也称为控制器，而数据部分一般称为结构（architecture）或数据路径。

在同步系统中，我们希望刚好在有效时钟沿到来时，所有变量发生改变，但是在实际电路中不一定发生。现代集成电路（IC）的尺寸等于或小于 0.1 μm，而现代微处理器的时钟频率往往为几个 GHz。在这些芯片中，线延迟时间与时钟周期同一个数量级。甚至对于两个使用统一时钟的

触发器来说，由于存在不同的线延迟，所以时钟沿到达这两个触发器的时间也不同。如果在不同部件的时钟分支中使用不同数量的组合逻辑电路，就会引起不同长度的时间延迟，并导致各个部分的时钟到来的时刻有着稍微的不同。这个现象称为**时钟偏移**。

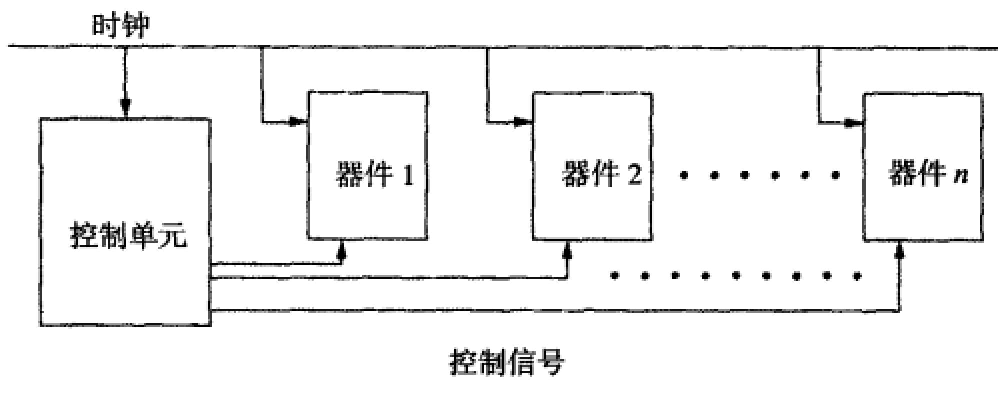


图 1.40　一个同步数字系统

另外，控制信号中的毛刺也可以引发一些问题。考虑图 1.41 给出的数字系统，这个系统的每个部件在时钟下降沿发生状态改变。几个触发器也在时钟下降沿到来时改变状态，而每个触发器状态改的时间，则取决于其自身的传输延迟。控制部分中触发器状态的改变要经过生成控制信号的组合逻辑电路中，最后改变响应的控制信号。这些控制信号的变化时间，则决定于生成控制信号的门电路的传输延迟和触发器的传输延迟。因此，时钟的下降沿到来后存在不确定时间段，在这段时间内控制信号可能发生改变。由于存在竞争冒险，所以控制信号中可能存在脉冲波形干扰、信号尖峰和毛刺。此外，当信号在电路的一个部分产生变化时，在电路的另一端会产生噪声。如图 1.41 的阴影所示，在每个时钟的下降沿之后的一小段时间内，在控制信号(*CS*)中可能产生噪声，而且控制信号的准确变化时间是无法知道的。

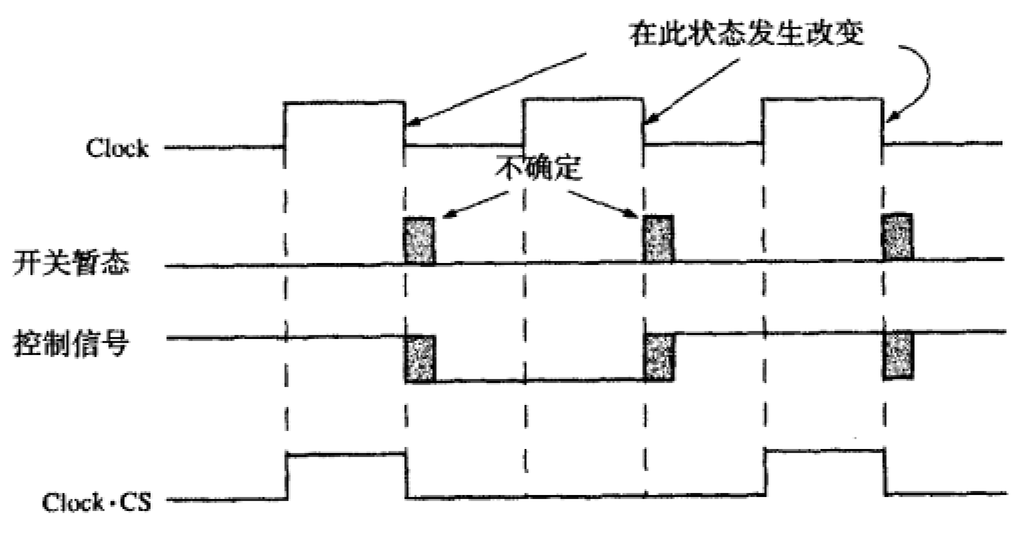
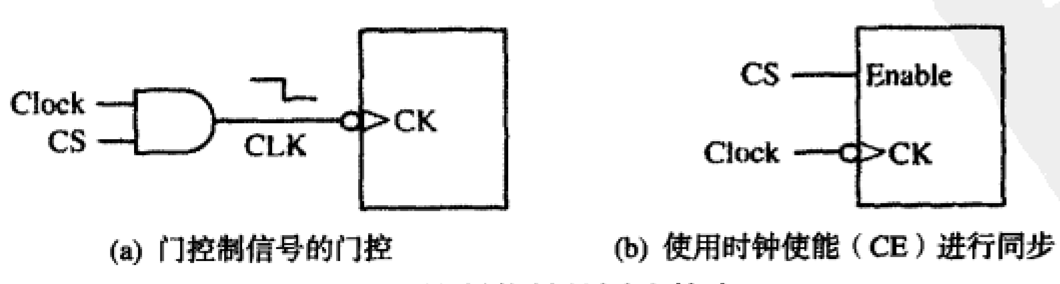


图 1.41　下降沿触发系统的时序图

如果对在数据部分中的部件要求只在脉冲下降沿并且在控制信号 $CS = 1$ 时改变状态，我们可以把时钟信号和控制信号 *CS* 做逻辑与运算，如图 1.42(a)所示。这一技术称为时钟门控技术。状态的转换除了与门中的一小段的延迟以外，其变化和时钟 CLK 是同步的。因为在 *CS* 中开关转换期间时钟是 0，所以门控的 CLK 信号是干净的。



(a) 门控制信号的门控　　(b) 使用时钟使能（CE）进行同步

图 1.42　控制信号的同步技术

如图 1.42(a)所示，用控制信号门控时钟信号，可以解决很多同步问题。然而，时钟门控技术在高速电路中也可能引入时钟偏移和其他时序问题。在实际应用中，我们更倾向于采用具有时钟使能管脚（*CE*）的部件，并用控制信号来控制该管脚，而不是用控制信号来门控时钟信号，如图 1.42(b)所示。用于同步系统中的很多寄存器、计数器和其他部件均具有使能输入端。当 enable = 1 时，部件对时钟进行响应；当 enable = 0 时，状态不发生改变。使能输入的使用避免了时钟门控用的门电路，同时也回避了相关的时序问题。

我们不鼓励设计者使用门电路控制时钟，或者把组合逻辑电路的输出反馈到时钟输入端。在某种程度上，虽然线延迟造成的时钟偏移是不可避免的，但是由组合逻辑电路造成的时钟偏移是可以避免的。为了使时序问题最小化，要尽可能少用如图 1.43 所示的电路。

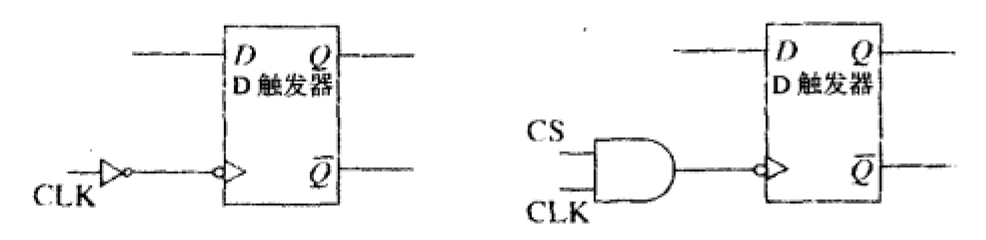


图 1.43　应避免使用的电路

由于线延迟和其他未知问题的存在，我们设计的最终电路中，有时无法避免时钟沿到达不同触发器的时刻是不同的。请看图 1.44，时钟到达两个触发器的时间有一点差异。好的同步操作意味着两个触发器是在同一个时钟下工作的。尽管第二个触发器的时钟存在延迟，但是它的状态变化必须在 Q_1 的新值到达 D_2 前进行触发。同步工作的最大时钟频率的确定也应考虑这两个时钟之间的延迟。

如果部件都没有使能端，并且同步只能靠门控制时钟，则我们必须要特别注意对每个时钟的正确门控。如果某一部件是在时钟下降沿触发的，则可以把时钟信号同控制信号做 AND 运算以实现门控，如图 1.42(a)所示。在下面的几段中，我们将讨论对时钟上升沿触发的部件的门控问题。

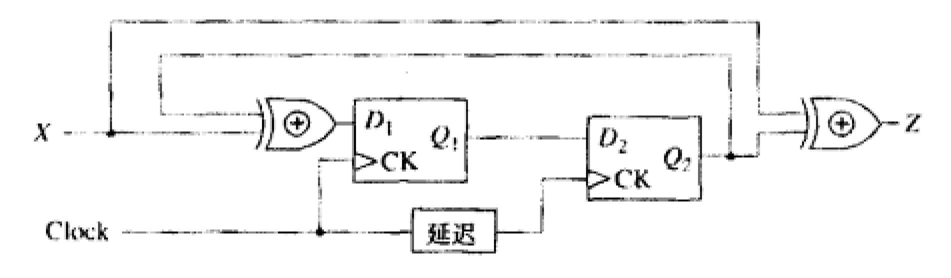


图 1.44　具有时钟偏移的电路

图 1.45 给出了一个数字系统的工作过程，这一系统是由时钟上升沿触发状态改变的部件组成的。这时引起噪声和不确定状态的开关暂态是在时钟的上升沿到来时形成的。图 1.45 中阴影部分表示控制信号 *CS* 有干扰的时间区域。如果我们期望该系统在 *CS* = 1 且时钟的上升沿到来时改变状态，那么在(a)和(c)时刻会发生状态转换，而在(b)时刻，由于 *CS* = 0 即使时钟上升沿到来也不改变状态。为了生成一个门控时钟信号，我们可能想到把时钟和控制信号做逻辑 AND 运算，如图 1.46(a)所示，结果是输入到 *CK* 端的信号可能是有干扰的并且有时序问题的。特别，CLK1 脉冲在点(a)是很短且有干扰的，它可能是太短导致不能触发，或者它可能有干扰的，从而导致多次地触发装置。通常在这种时钟下系统的同步得不到保障，因为控制信号的改变依赖于控制电路中的一些触发器的状态转换。当然，图 1.46 中点(b)的时钟上升沿不应触发，但它也有干扰。更糟糕的是，在点(b)本不应触发的，但也有可能在点(b)的附近触发。这时因为时钟上升沿的时候 *CS* = 0，所以直到下一个上升沿到来时，当 *CS* = 1 时，触发才会发生。

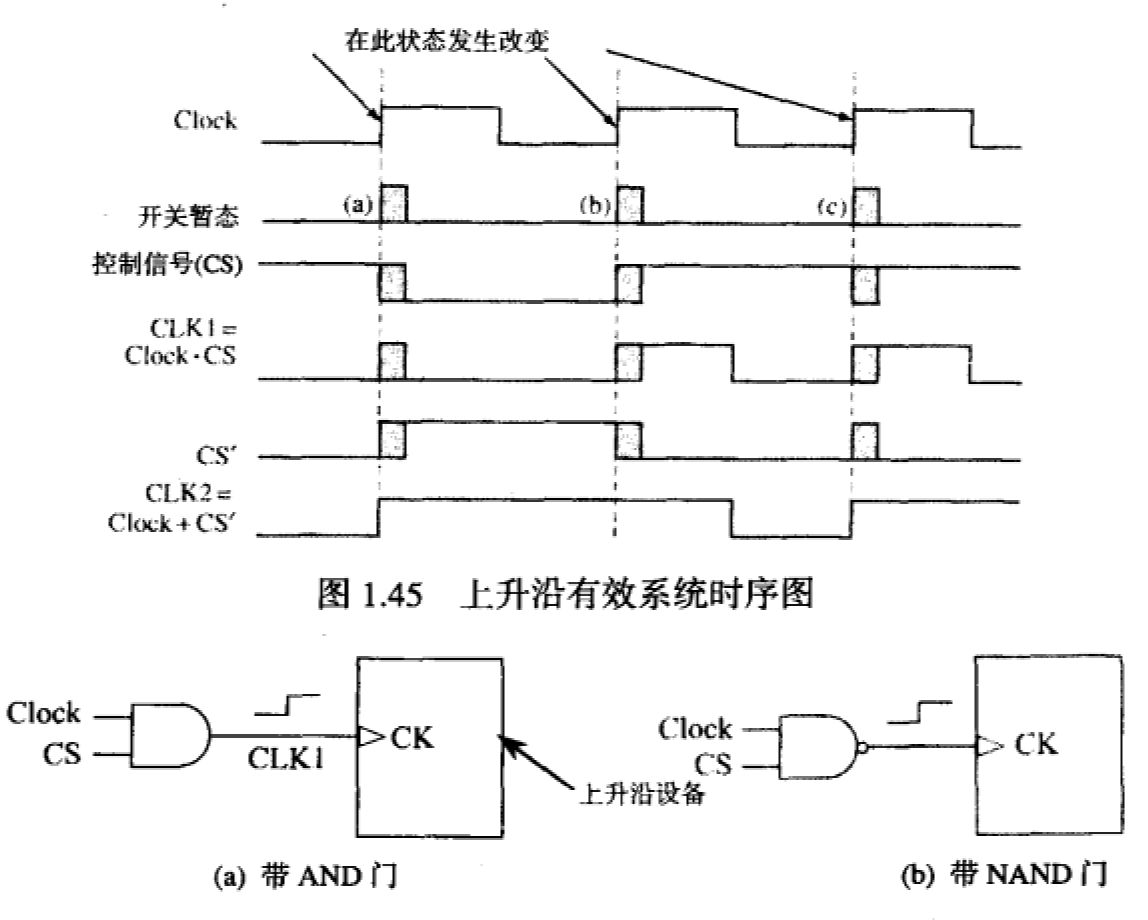


图 1.45　上升沿有效系统时序图

(a) 带 AND 门　　(b) 带 NAND 门

图 1.46　不正确的时钟门控（对上升沿有效的部件）

对于一个在时钟上升沿触发的部件，如果我们把图 1.42 中的 AND 门变为图 1.46(b)中的 NAND 门，由于同步发生在错误的时钟沿，所以它会出现错误。图 1.47 给出了生成门控时钟信号的一个正确方法，只有当控制信号为正，且时钟上升沿将要到来时，我们把 *CK* 端输入正好有一个上升沿。*CK* 输入为

$$CLK2 = (CS \cdot clock')' = CS' + clock$$

图 1.45 中的最后一个波形给出了这一门控时钟信号。虽然该电路可以解决同步问题，但是设计者还是尽量少用此电路为好。

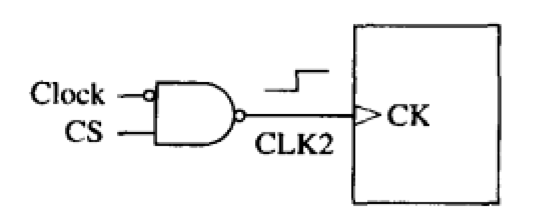


图 1.47　正确的时钟门控（对上升沿有效的部件）

总之，同步设计遵循以下原则：

- 方法：所有输入到触发器、寄存器和计数器等部件的时钟，必须是由系统时钟直接驱动的。
- 结果：所有的状态变化必须由时钟的有效沿（上升沿或下降沿）触发。
- 优势：所有开关暂态、开关噪声和其他干扰都发生在时钟脉冲间，对系统性能没有影响。

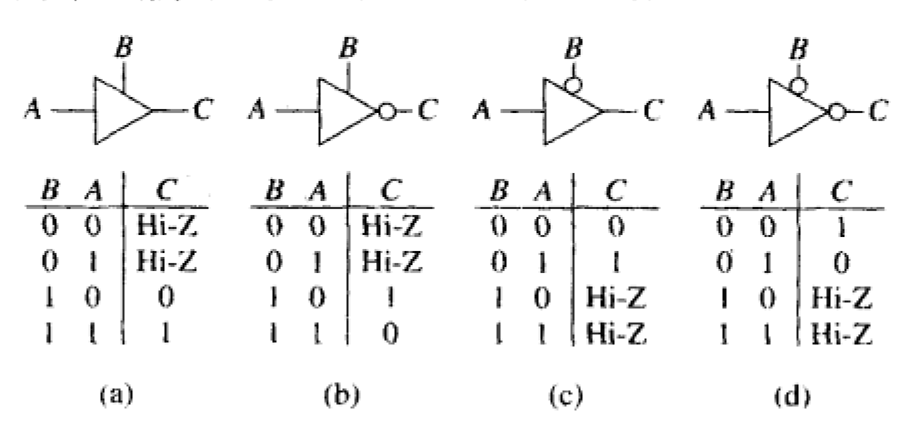
异步设计通常比同步设计更难。因为没有一个时钟用于状态转换的同步，当需要一些状态变量同时改变时，就可能出现问题，并且当最终状态取决于这些变量的变化次序时会出现竞争问题。异步设计需要特殊的技术以消除竞争和冒险。另一方面，同步设计存在一些缺点：在高速的电路中，线延时问题特别突出，时钟信号的布线一定要小心谨慎，要保证所有的时钟输入同时到达每个部件（就是使时钟偏移最小）。系统的最大时钟频率由最长路径的最大延迟决定。由于系统输

入可能不与时钟同步，所以同步器的使用是必要的。同步系统的耗电要比异步系统更大，这是因为同步芯片中的时钟分布电路消耗的能量通常占芯片消耗能量的大部分。

1.11 三态逻辑和总线

通常，如果我们把两个门或触发器的输出连接起来，电路将不能正常工作，并且这会对电路造成损坏。因此，当我们需要把多个门的输出连接到同一条线或同一个通道时，为了解决上面的问题就要用到三态缓冲器。除了逻辑高电平和逻辑低电平外，三态缓冲器还具有高阻状态（Hi-Z），高阻状态等效于把电路断开。在数字系统中，时常需要在一些系统部件之间来回地传递数据。三态总线可以用于寄存器之间的数据传输。当多个门连接在一个线上时，我们希望只有一个门用于驱动该线，而其他的门不起作用就像没有连接到该线一样。我们可以通过高阻状态达到此目的。

三态缓冲器的输出可以是取反的，也可以是不取反的。控制输入可以是高电平有效，也可以是低电平有效。图 1.48 说明了三态缓冲器的四种类型。*B* 是控制输入端，决定缓冲器能否输出数据。当一个缓冲器被允许的时候，输出(*C*)等同于输入(*A*)或者(*A*)的取反。然而，如果在某一时刻只有一个输出是允许的话，我们可以把两个三态缓冲器的输出端连在一起。



(a)

B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

(b)

B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	1
1	1	0

(c)

B	A	C
0	0	0
0	1	1
1	0	Hi-Z
1	1	Hi-Z

(d)

B	A	C
0	0	1
0	1	0
1	0	Hi-Z
1	1	Hi-Z

图 1.48 4 种三态缓冲器

图 1.49 是一个连有三个寄存器的三态总线系统。每一个寄存器均是 8 位的，而且总线也是由并行执行的 8 根线组成的。图中每个三态缓冲器的符号代表并联的 8 个缓冲器，它们共享一个使能端。该系统总是只有一组缓冲器被允许。例如，当 *Enb* =1，寄存器 B 的输出被送到总线。总线上的数据再传输到寄存器 A、寄存器 B 和寄存器 C 的输入端。但是，只有在 *load*=1 和有效时钟沿到来时，寄存器才会存入数据。因此，当 *Enb* = *Ldc* = 1 时，在时钟上升沿，B 寄存器中的数据将会复制到 C 寄存器。如果，*Eni* = *Lda* = *Ldb* = 1，则在时钟上升沿，寄存器 A 和 B 都会存入输入数据。

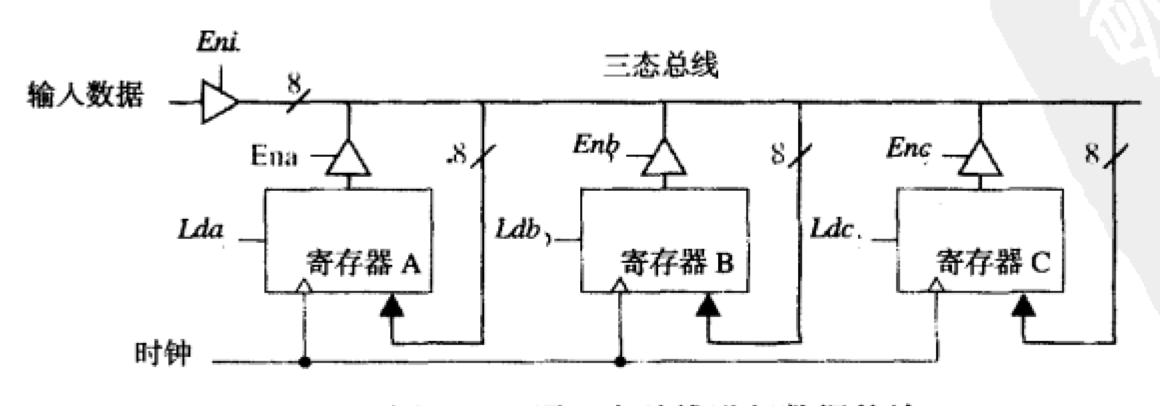


图 1.49 用三态总线进行数据传输

习题

1.1 写出下面等式的真值表：

$$F=(A\oplus B)\cdot C+A'\cdot(B'\oplus C)$$

1.2 写出全减器的真值表，并分别写出 *Diff* 和 B_{in} 的与或式和或与式。全减器是可以计算三个数 X, Y, B_{in} 的差的，即 $Diff=X-Y-B_{in}$。当 $X<Y+B_{in}$ 时，借位输出 B_{out} 置位。

1.3 使用 4 变量的卡诺图化简 Z。其中 *ABCD* 为控制电路的状态，并假设电路不存在 0100, 0001, 1001 这三个状态。

$$Z=BC'DE+ACDF'+ABCD'F'+ABC'D'G+B'CD+ABC'D'H'$$

1.4 使用含有嵌入变量的 4 变量卡诺图，把下列各式化简为最简与或式。(a)和(b)中的 m_i 表示变量 A、B、C 和 D 的最小项。

(a) $F(A,B,C,D,E)=\sum m(0,4,6,13,14)+\sum d(2,9)+E(m_1+m_{12})$

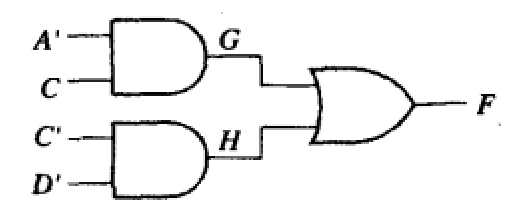
(b) $Z(A,B,C,D,E,F,G)=\sum m(2,5,6,9)+\sum d(1,3,4,13,14)+E(m_{11}+m_{12})+F(m_{10})+G(m_0)$

(c) $H=A'B'CDF'+A'CD+A'B'CD'E+BCDF'$

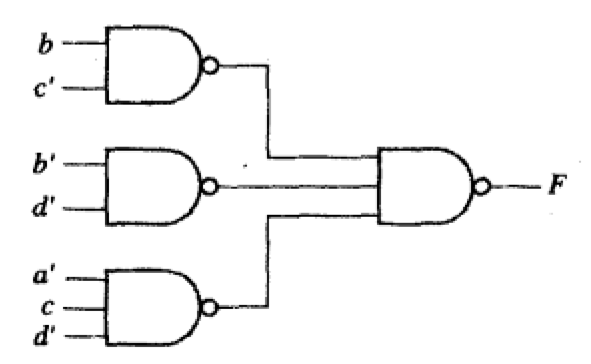
(d) $G=C'E'F+DEF+AD'E'F'+BC'E'F+AD'EF'$

提示：卡诺图中哪些变量应该作为输入变量，哪些应该作为嵌入变量呢？

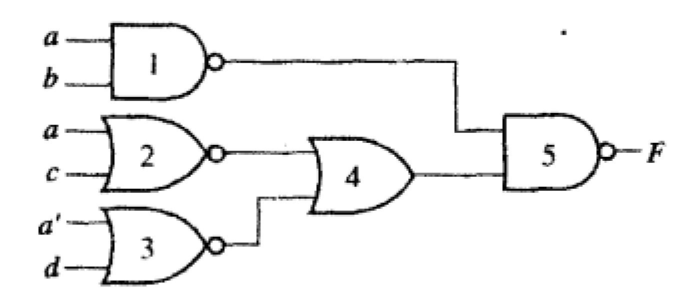
1.5 判断下面电路中是否存在静态 1 冒险。列出每一个冒险出现的条件，画出冒险出现时各个事件的时序图[与图 1.10(b)相似]。



1.6 找出所给电路中存在的所有 1 冒险，并指出如果要去掉这些冒险，那么必须把电路做哪些改变？



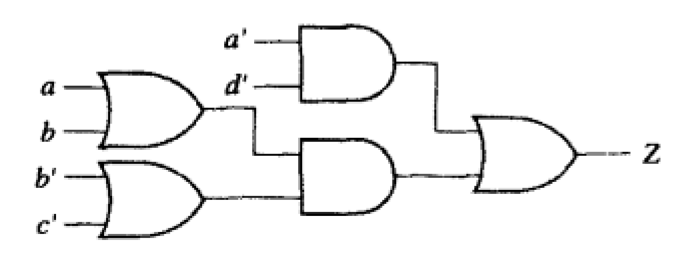
1.7 (a) 找出下面电路图中存在的静态冒险，指出每一个冒险出现时各输入变量的值及其变化。针对其中的一个冒险，指出当其发生时各门电路输出的变化次序。



(b) 用于与非电路设计实现具有相同逻辑方程的电路，要求该电路不存在静态冒险。

1.8 (a) 找出下面电路图中存在的静态冒险，并列出存在冒险的各种情况。

(b) 重新设计此电路，要求此电路不存在静态冒险，可以用最多有 3 个输入的门电路。



1.9 (a) 试用一个 J-K 触发器组成一个 T 触发器。

(b) 试用一个 D 触发器和一些门电路组成一个 J-K 触发器。

1.10 使用 2 个 D 锁存器和一些门电路构造一个时钟(*CLK*)上升沿触发的 D 触发器。在下面的时序图中，Q_1 和 Q_2 为锁存器的输出，完成下图并验证触发器输出值在时钟上升沿变为与 *D* 相同的值。

1.11 在具有一个输入和一个输出的同步时序电路中，如果输入序列为 0101 或 0110，则输出为两个连续的 1，第一个 1 与输入信号 0101 或 0110 的最后一位同步输出，而且当第二个 1 输出后系统重新复位。例如，

输入序列 X = 010011101010 101101…
输出序列 Z = 000000000011 000011…

(a) 画出具有最少状态（6 个状态）的 Mealy 状态图和状态表。

(b) 试用一个较好的状态赋值，先用 J-K 触发器和 NAND 门实现该电路，再改用 NOR 门实现该电路（此部分用手画）。

(c) 用 LogicAid 程序检验(b)中的结果，并使用该程序给出基于 NAND 门的其他两种赋值方法的结果。

1.12 一个时序电路具有一个输入(X)和两个输出(Z_1, Z_2)。当输入为 010 时，输出 $Z_1 = 1$；当输入为 100 时，输出 $Z_2 = 1$。并且一旦 $Z_2 = 1$ 时，Z_1 绝不为 1。

(a) 画出具有最少状态（8 个状态）的 Mealy 状态图和状态表。

(b) 试用一个较好的状态赋值，先用 J-K 触发器和 NAND 门实现该电路，再用 NOR 门实现该电路（此部分用手画）。

(c) 用 LogicAid 程序检验(b)的答案，并使用该程序给出基于 NAND 门的其他两种赋值方法的结果。

1.13 一个时序电路有一个输入(X)和两个输出（S, V）。X 表示一个 4 位二进制数 N，且从最低有效位开始输入。S 表示一个 4 位二进制数 $N+2$，且从最低有效位开始输出。当第 4 个二进制位输入时，如果 $N+2$ 要用多于 4 位二进制数表示，则 $V=1$，否则 $V=0$，此时 S 的取值均不是任意的。当收到 X 的第 4 位后，电路复位。

(a) 画出具有最少状态（6 个状态）的 Mealy 状态图和状态表。

(b) 试用一个较好的状态赋值，并且先用 J-K 触发器和与非门实现该电路，再用 NOR 门实现该电路（此部分用手画）。

(c) 用 LogicAid 程序检验(b)的答案，并使用该程序给出基于 NAND 门的其他两种赋值方法的结果。

1.14 一个时序电路有一个输入(*X*)和两个输出（*D*, *B*）。*X* 表示一个 4 位二进制数 *N*，且从最低有效位开始输入。*D* 表示一个 4 位二进制数 *N*–2，且从最低有效位开始输出。当第 4 个二进制位输入时，如果 *N*–2 为负，则 *B* = 1，否则 *B* = 0，此时 *S* 的取值均不是任意的。当收到 *X* 的第 4 位后，电路复位。

(a) 画出具有最少状态（6 个状态）的 Mealy 状态图和状态表。

(b) 试用一个较好的状态赋值，并且先用 J-K 触发器和 NAND 门实现该电路，再用 NOR 门实现该电路（此部分用手画）。

(c) 用 LogicAid 程序检验(b)的答案，并使用该程序给出基于 NAND 的其他两种赋值方法的结果。

1.15 在具有一个输入和一个输出的 Moore 时序电路中，如果输入序列为 111，则输出为 1；如果输入序列为 000，则输出为 0。在其他时间，输出均保持前一时刻的值。例如，

$$X=0\,1\,0\,0\,1\,1\,1\,0\,1\,0\,0\,0\,1\,1\,1\,0\,0\,1\,0\,0\,0$$
$$Z=0\,0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,1\,0\,0\,0\,1\,1\,1\,1\,1\,1\,0$$

请给出该电路的 Moore 状态图和状态表。

1.16 请给出模 6 计数器的状态转移表和触发器的输入表达式。模 6 计数器要求可以从 000 数到 101，然后重新开始计数，触发器要求使用 J-K 触发器。

1.17 一个计数器可以从 1 数到 6，然后再重新开始计数。请给出该计数器的状态转移表和 D 触发器的输入表达式。

1.18 把下面状态表化简，使状态数最小。

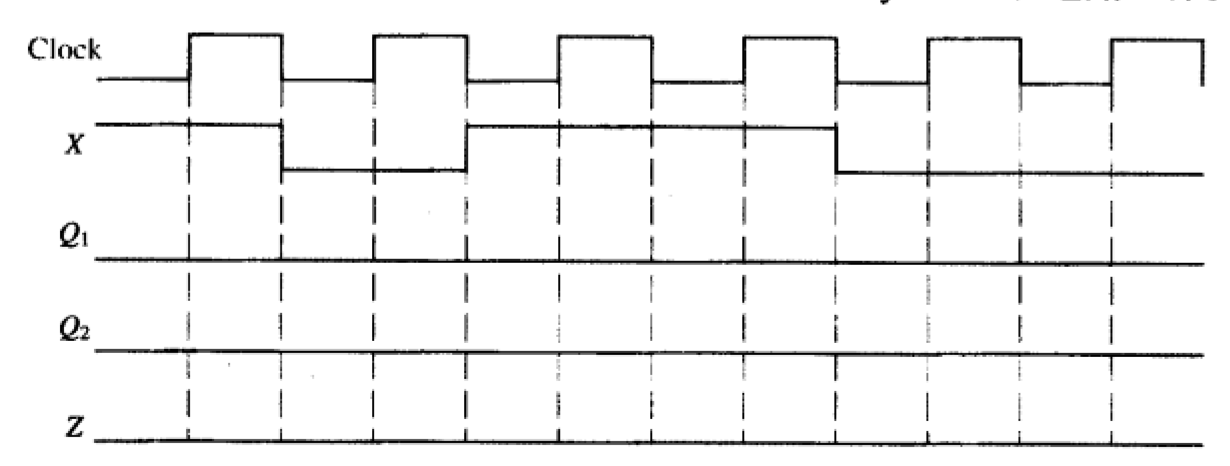
当前状态	下一状态		输出	
	X = 0	*X* = 1	*X* = 0	*X* = 1
A	*B*	*G*	0	1
B	*A*	*D*	1	1
C	*F*	*G*	0	1
D	*H*	*A*	0	0
E	*G*	*C*	0	0
F	*C*	*D*	1	1
G	*G*	*E*	0	0
H	*G*	*D*	0	0

1.19 一个 Mealy 时序电路由图 1.44 电路来实现，当输入 *X* 变化时，该电路在时钟下降沿到来时发生改变。

(a) 完成下面的时序图。指出读出输出(*Z*)的最佳时间。假设图中 delay = 0 ns，触发器和 XOR 门的一般传输延迟都为正常值 10 ns，时钟周期为 100 ns。

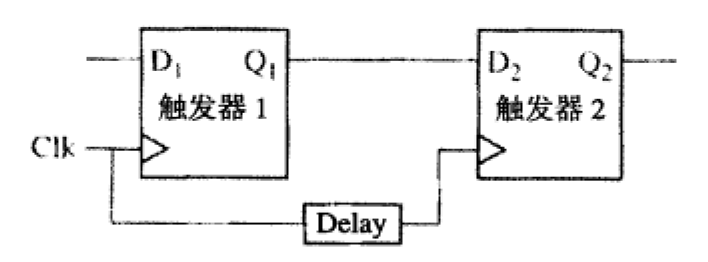
(b) 设下面的延迟：XOR 门延迟为 10 ~ 20 ns，触发器的传输延迟为 5 ~ 10 ns，建立时间为 5 ns，保持时间为 2 ns，图中 delay = 0 ns，计算正常同步工作时的最大时钟频率。要求考虑两条反馈路径，包含触发器传输延迟的反馈路径和 *X* 改变时就开始的路径。

(c) 假设时钟周期为 100 ns，与(b)中的所有时序参数一样。试给出该电路仍同步工作的 delay 的最大延迟？即该电路的状态序列必须与无延迟（delay = 0 ns）电路一样。



1.20 两个触发器按下图方式连接。两个时钟输入之间的连线延迟会导致时钟偏移，这可能会使两个触发器不同步。触发器从时钟有效沿到 Q 的传输延迟为 $10\text{ ns} < t_p < 15\text{ ns}$；建立时间和保持时间分别为 4 ns 和 2 ns。

(a) 图中延迟最大为多少时电路仍可以正常同步工作？画出时序图验证你的答案。



(b) 假设延迟<3 ns，最小允许时钟周期为多少？

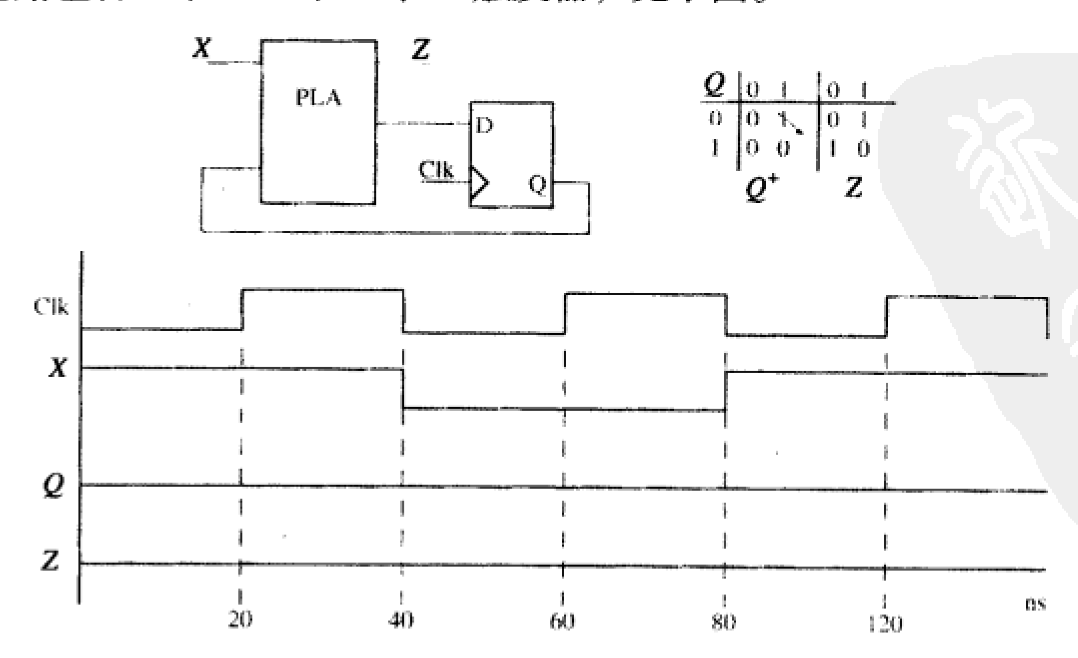
1.21 一个 D 触发器从时钟沿到 Q 的传输延迟为 7 ns，其建立时间和保持时间分别为 10 ns 和 5 ns。触发器输入时钟的周期为 50 ns（前 25 ns 前为低电平，25 ~ 50 ns 之间为高电平，依次类推）。假设门延迟在 2 ~ 4 ns 之间，触发器在上升沿触发。

(a) 假设输入 D 从 0 ~ 10 ns 为 0，从 10 ~ 35 ns 为 1，从 35 ~ 70 ns 为 0，之后一直为 1。画出时钟、D 和 Q 的时序图（画到 100 ns）。如果输出不能确定（由于不满足建立时间和持保持时间），则在时序图中用 XX 标示。

(b) 触发器输入 D 在时钟沿到来前_____ns 和在时钟沿到来后_____ns 不能发生改变。

(c) 外部输入在时钟沿到来前_____ns 和在时钟沿到来后_____ns 不能发生改变。

1.22 一个时序电路包含一个 PLA 和一个 D 触发器，见下图。



(a) 完成下面的时序图。假设 PLA 的传输延迟为 5 ~ 10 ns，从有效时钟沿到 D 触发器输出的传输延迟为 5 ~ 10 ns，在时序图中用箭头指出 Q 和 Z 允许改变的时间间隔，并计算传输延迟的范围。

(b) 假设 X 总是在时钟下降沿到来时改变，求触发器可正常工作的最大允许建立时间和保持时间。

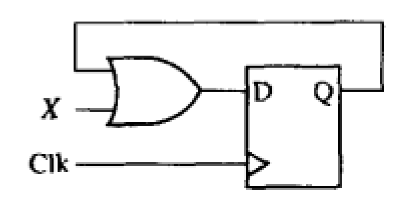
1.23 一个 D 触发器从时钟沿到 Q 的传输延迟为 15 ns。触发器的建立时间为 10 ns，保持时间为 2 ns。触发器输入时钟的周期为 50 ns（在 25 ns 前为低电平，25 ~ 50 ns 之间为高电平，依次类推）。触发器在上升沿触发。D 在 20 ns 时变为高电平，在 40 ns 时变为低电平，在 60 ns 时变为高电平，在 80 ns 时变为低电平，依次类推。画出时钟、D 和 Q 的时序图（画到 100 ns）。如果输出不能确定（由于不满足建立时间和保持时间），则在时序图中用 XX 标示。

1.24 一个 D 触发器的建立时间为 5 ns，保持时间为 3 ns，从时钟上升沿到触发器输出改变之间的传输延迟的取值范围为 6 ~ 12 ns。或门的延迟取值范围为 1 ~ 4 ns。

(a) 求该电路正常工作所允许的最小时钟周期。

(b) 求在时钟上升沿之后 X 允许改变的最早时间。

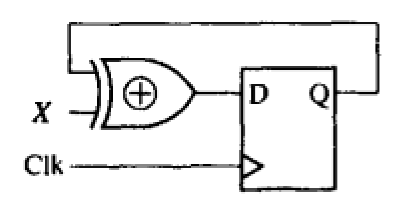
(c) 如何用一个 J-K 触发器构建一个 T 触发器。用框图进行说明，触发器内部电路不用考虑。



1.25 一个 D 触发器的建立时间为 4 ns，保持时间为 2 ns。从时钟上升沿到触发器输出改变的传输延迟为 2 ~ 16 ns，异或门的延迟为 12 ~ 24 ns。

(a) 求该电路正常工作所允许的最小时钟周期。

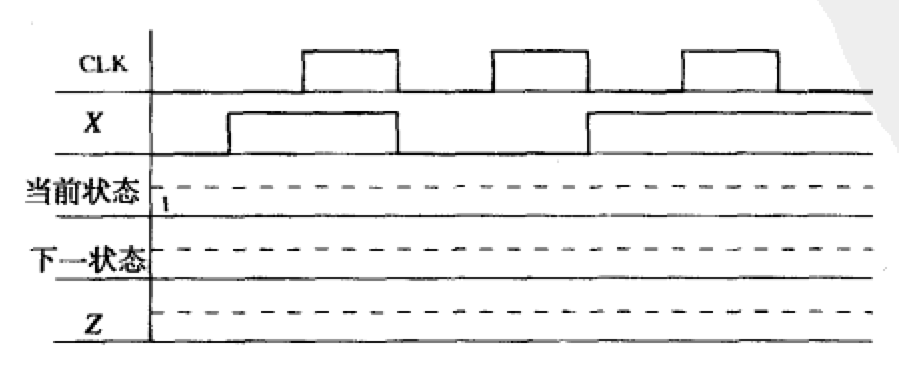
(b) 求在时钟上升沿之后 X 允许改变并能保持同步的最早和最晚时间[假设时钟频率为(a)中 结果]。



1.26 一个 Mealy 时序电路的状态表如下：

当前状态	下一状态 $X=0$	下一状态 $X=1$	Z $X=0$	Z $X=1$
1	2	3	0	1
2	3	1	1	0
3	2	2	1	0

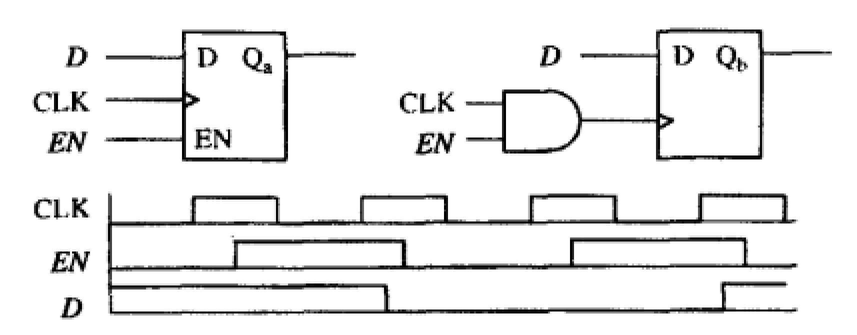
完成下面的时序图，并在图上标注出何时应该对 Z 的值进行读操作。所有的状态改变均发生在时钟上升沿。



1.27 (a) 下面两电路的时序图相同吗?

(b) 画出 Q_a 和 Q_b 的时序图。

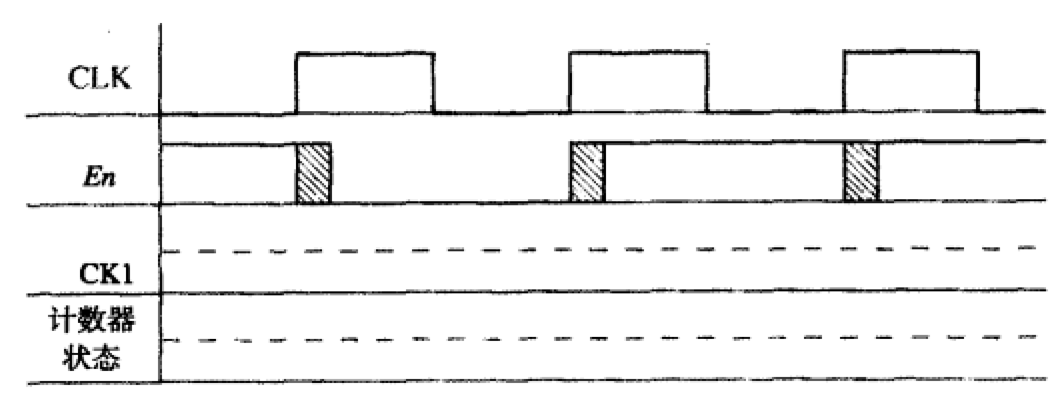
(c) 如果你对(a)的答案是否定的，则要使两电路有相同的时序图，那么第二个电路应做何改动？（不改变触发器）



1.28 一个简单的二进制计数器只有一个时钟输入（CK1）。此计数器在 CK1 的上升沿加 1。

(a) 连续信号 *En* 和系统时钟（CLK），使之当 *En* = 1 时，计数器在 CLK 的上升沿加 1；当 *En* = 0，则计数器状态不改变。

(b) 完成下面的时序图，并用根据时序图解释为何 *En* 在时钟上升沿后的过渡区域没有给计数器的操作带来影响。



1.29 根参考图 1.49，试给 *Eni*, *Ena*, *Enb*, *Enc*, *Lda*, *Ldb*, *Ldc* 赋值，使之在有效时钟沿把寄存器 Reg.C 中的数据复制到寄存器 Reg.A 和寄存器 Reg.B 中?

第 2 章　VHDL 简介

随着现代集成电路技术的发展，在一个芯片上可以集成越来越多的元件，数字系统也变得更加复杂。以前，我们认为一个集成电路包含很多晶体管是一个奇迹，但是现在随着科技的进步，我们已经开始在超大规模集成电路（VLSI）上取得进展。早期的集成电路按集成程度可分为三类：小规模集成电路 SSI、中等规模集成电路 MSI 和大规模集成电路 LSI。SSI 是指一个 IC 上集成 1～20 个门的集成电路；MSI 是指在一个 IC 上集成 20～200 个门的集成电路；LSI 是指在一个 IC 上集成 200 到几千个门的集成电路。很多常用模块，像加法器、乘法器、译码器、寄存器和计数器等都属于 MSI 集成电路。VLSI 是指一个芯片上集成 10 000 个门的集成电路。这种不同集成电路之间的界限，今天已经变得很模糊了。现在很多微处理器都包含 1 亿多个门。如果按原先 VLSI 集成电路规模来说的话，今天的集成度应该称为特大规模集成电路了（ULSI）。尽管集成度上发生了变化，还有分类定义上的模糊，但是 VLSI 仍用于指现在的集成电路，而 LSI 却基本不用了。

由于数字系统变得更加复杂，在门电路和触发器层次上进行系统细节的设计就变为十分烦琐和耗时。20～30 年前，数字系统设计还可以通过手工画图和在面包板上布线而完成，而现在的硬件设计通常就不采用这种手工方式了。

本章中，我们先介绍计算机辅助设计，然后再介绍硬件描述语言 VHDL。我们主要介绍它的基本特征，并用实例说明如何使用 VHDL 语言描述、模拟和综合数字系统。VHDL 语言的高级功能将在第 8 章中介绍。

2.1　计算机辅助设计

在过去的十年里，计算机辅助设计（CAD）工具得到得到了极大的发展。现在有各种工具软件可以用于数字系统的设计。我们都可以不用个别的元件和连线，就能完成数字电路原型甚至是最终设计。

图 2.1 给出了现代数字系统的设计流程。与所有工程设计相同，设计流程的第一步是规范问题，即从列出设计要求到设计规范。下一步是在概念层面上的顶层设计，即从框图层面或算法层面进行设计。

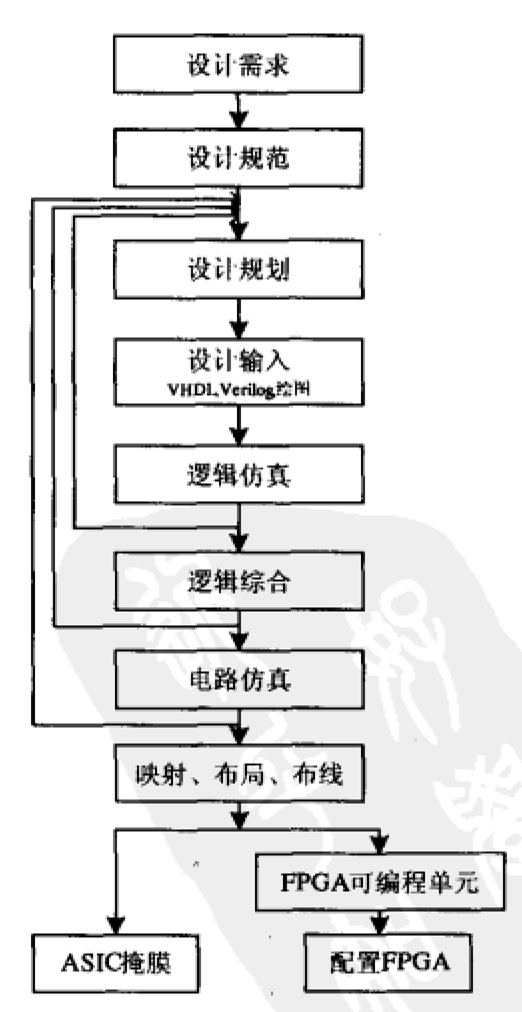


图 2.1　现代数字系统设计流程

下一步就是设计输入了。以前是靠手工绘图或图纸。现在有了 CAD 工具，就可以把前一步中的顶层设计以适当的形式输入到 CAD 工具中。早在 CAD 工具主要用于设计画图，称为设计图采集。绘图编辑器通常带有标准数字模块库，像各种门电路、触发器、乘法器、译码器、计数器和寄存器等。ORCAD 公司就提供了非常普及的绘图编辑器。现在，我们使用硬件描述语言（HDL）进行设计。最常用的两种硬

件描述语言为 VHDL 和 Verilog。VHDL 意为超高速集成电路硬件描述语言，它是 VHSIC (very high speed integrated circuit) Hardware Description Language 的缩写。

硬件描述语言可以使数字系统在更高层面进行设计和调试，而不像设计图采集那样在门电路、触发器和标准 MSI 模块层面上设计。其实在设计之初，我们不必考虑具体的门电路和触发器结构。硬件描述语言的一个设计方法是行为描述，在这种行为描述方式设计中，我们只需给出系统在工作流程图或算法层面上的行为，无需顾及具体的物理部分、元件和电路实现问题。VHDL 和 Verilog 设计的另一种方法是结构描述设计，在这种结构描述设计中，我们将涉及每个具体的元件或者每个元件的具体实现问题。VHDL 和 Verilog 的结构描述设计模块可以看做是对连接具体的门电路和触发器的电路图的文字描述。

一旦完成一个设计之后，很重要的一步就是对所设计的系统进行仿真，以确定其功能的正确与否。最初的仿真应该是在较高的行为描述层面上的仿真，它可以找到设计中的最初问题。如果出现问题，设计者应该回到设计规划，改变设计并使其满足设计要求。

如果通过仿真验证了系统的功能，则下一步就是逻辑综合。逻辑综合是指"把较高层面上设计的系统转化为在门电路和触发器层面上实现的时间逻辑电路"。现在许多计算机辅助设计工具都能完成这种转化。综合工具的输出为一个网表，给出了逻辑门列表和这些逻辑门的连接关系表。逻辑综合很像把高级语言程序（比如 C 语言），通过编译器把它转化为机器语言程序。C 语言编译器既能生成最优化的机器码，也能产生非优化的机器码。同样一个综合工具也能生成优化的硬件实现或者非优化的硬件实现。这是由于这些综合工具所用的转化算法和优化技术的不同所导致的。综合工具其实就是一个把设计描述转换为硬件的编译器，只不过是借助于"综合"的封装来表示与设计编译器、芯片编译器等相同的意思。

设计流程的下一步是电路仿真（综合-后仿真）。在前面的较高层面上的逻辑仿真中，不能考虑该系统的具体硬件电路中的元器件。如果在电路仿真中出现问题，那么设计者需要重新修改设计以满足时序要求。要完成实现一个比较满意的硬件实现是一个反复修改过程。

下一步，设计者就可以进行芯片的具体实现。可供选择的技术有很多（如图 2.2 所示），我们可以选用完全定制的 IC，也可以根据厂家提供的标准模块来实现。

在图 2.2 中，从密度和费用来说，最底层是老式电路板（焊有常用门、触发器和其他标准逻辑模块）。在密度上稍高一点的是可编程逻辑阵列（PLA）、可编程阵列逻辑（PAL）和简单可编程逻辑器件（SPLD）。具有更高的密度和更多门的 PLD 是复杂可编程逻辑器件（CPLD）、然后是常见的现场可编程门阵列（FPGA）和掩模可编程门阵列（MPGA），一般统称为门阵列。集成度最高和性能最好的是定制专用集成电路（ASIC）。

现在最常用的两种芯片实现目标技术为 FPGA 和 ASIC。对于这两种技术来说，设计流程的前几个步骤（设计前端）都大体相同，但是最后几个步骤（设计后端）中实行不同的操作，如图 2.1 所示。在设计的后端，首先确定最后芯片实现技术，然后把设计映射布局到目标 FPGA 或 ASIC 上的具体的部分。连接各个组成部分的连线路径由布线确定。如果设计一个 ASIC 芯片，则由设计的布线生成一个光掩模（photomask），它将在 IC 制造过程中使用。如果用 FPGA 实现，那么设计就转换为一种格式，它决定了对 FPGA 中哪个可编程的点进行怎样的操作。在现代 FPGA 中，编程只需在 FPGA 的可编程单元中写入 01 序列即可，这样除了个人电脑（PC）之外，就不需要其他的编程设备了。

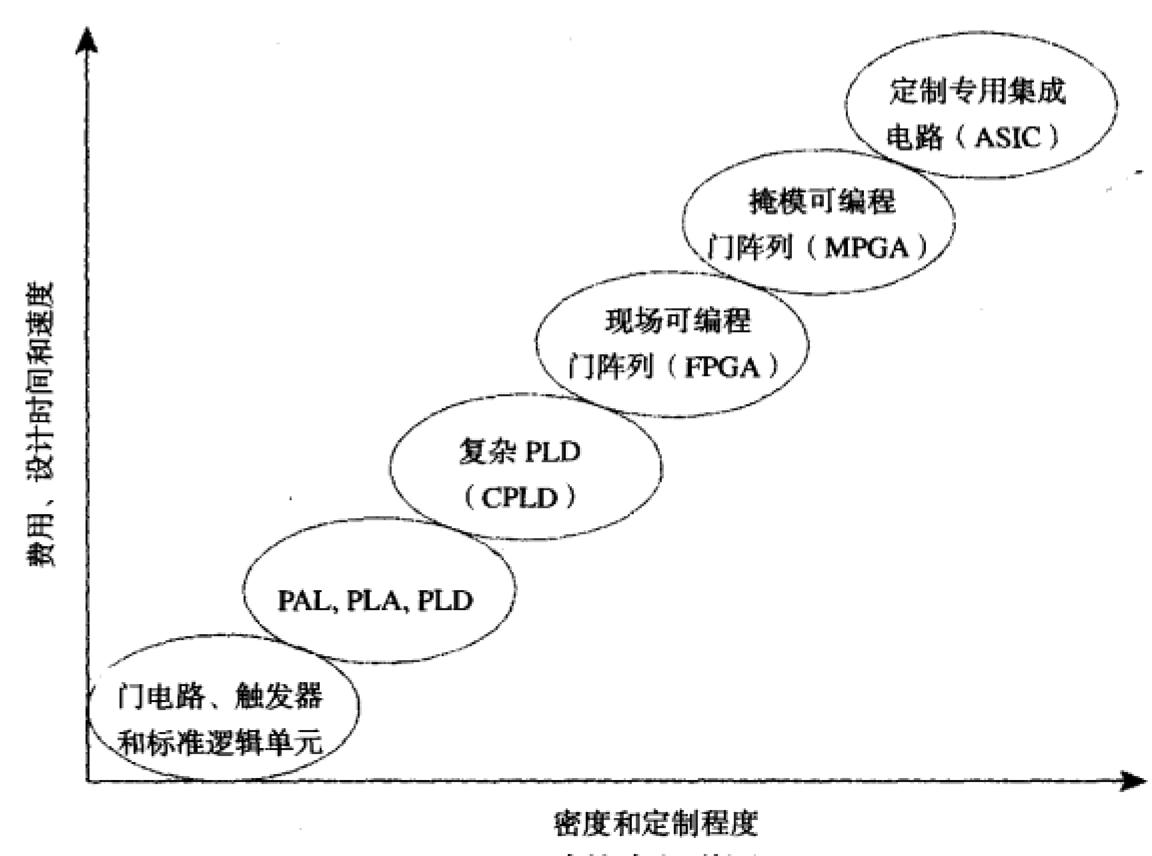


图 2.2　设计技术频谱图

2.2　硬件描述语言

现在用硬件描述语言（HDL）进行设计是一种流行趋势。VHDL 和 Verilog 是两种常用的硬件描述语言。本书主要介绍如何用 VHDL 语言进行现代数字系统设计。

VHDL 是一种用来描述数字系统行为和结构的硬件描述语言，它是通用语言，所以广泛适用于描述和仿真各种数字系统，小到几个门，大到许多复杂集成电路相连的系统。VHDL 的起因是 DoD（美国国防部）资助的寻求数字系统统一描述的项目。当 VHDL 开发后，其主要目标是给出一种机理，使之硬件的描述和语言叙述清晰不含糊，从较高层面上的描述综合实现硬件并不是它的初衷。当 VHDL 成为美国电子电气工程师协会 IEEE 标准后，它才在工业中得到广泛的应用。1987 年 IEEE 建立了一个 VHDL 标准（VHDL—87），随后在 1993 年对该标准做了修正（VHDL—93），最后在 2000 年和 2002 年完成了更完善的 VHDL 标准。

VHDL 语言有三个不同层面上的数字系统描述方式，它们分别是行为、数据流和结构描述方式。例如，二进制加法器，在行为描述方式下就是一个完成把两个二进制数相加函数功能的系统，而不用给出任何具体硬件实现的细节。该加法器在数据流描述方式下，要给出组成该加法器的各种门电路和它们的相互连接。

自然引出自顶向下的设计方法，系统先在高层面上设计和仿真验证。在这一层面上调试通过后，系统可以逐步展开细化，直至最后达到接近实际硬件实现的结构描述。基于 VHDL 的设计与实现技术本身无关的，该设计可以用今天的技术来实现，也可以把它作为应用将来某新技术的设出发点。虽然最初 VHDL 只是用做一个硬件的文档描述语言，但是现在大多数 VHDL 主要用于仿真和逻辑综合。

Verilog 也是一种流行的硬件描述语言，它是与美国国防部投资开发 VHDL 大体相同时间，由工业界开发的。1984 年，Verilog 作为一种私用硬件描述语言，由 Gateway Design Automation 公司给出。1988 年，Synopsis 公司为 Verilog 开发了综合工具。1995 年，Verilog 成为 IEEE 的一个标准。

VHDL 的语法是基于 ADA 语言的，而 Verilog 的语法是基于 C 语言的。ADA 也是一种由 DoD 支持的通用编程语言。由于 Verilog 的语法是基于 C 语言的，所以一些人发现它更容易掌握。而许多人发现 VHDL 在设计和描述大型系统时具有突出优势。目前这两种语言各占一半市场份额。这两种语言都可以满足数字设计的大多数要求。通常，设计公司更倾向于使用它们熟悉的语言。因此，使用 Verilog 的公司就一直使用 Verilog 语言，使用 VHDL 的公司就一直使用 VHDL。只要你掌握了这两种语言中的一种，那么掌握另一种也并不困难。

最近，人们试图开发系统设计语言，比如 System C, Handel-C 和 System Verilog。System C 是 C++语言的一种扩展，所以对于一些熟练掌握通用开发软件的人来说，这种语言很好掌握。这些语言都是在较高层面描述大型数字系统为目标，基本上用于检测和验证系统。如果把一个大型系统分成几部分，每一部分由一个小组来完成，那么一个小组在自己的最初设计中，就可以在系统层面上用另一小组设计的模块。这样，一般只有在系统集成时候暴露的一些问题，可以在设计初期就会显露出来，缩短了大型系统的设计周期。系统层面上的仿真语言多用于设计大型系统。

2.2.1　如何学习一种语言

不论是要学习一门用于日常交流的语言（如英语、西班牙语和法语等），或是要学习一门计算机语言（如 C 语言），还是要学习一种特殊应用语言（如 VHDL），只要你是想学习一种新的语言，那么你就会面临许多挑战。如果你要学习的这门语言不是你所熟悉的，那么很自然地，你就会把它同你所熟悉的语言进行比较。对于 VHDL 语言来说，如果你之前学过其他的硬件描述语言，那么你可以把二者进行比较学习。但是，在与 C 语言进行比较时应该特别注意，因为 VHDL 和 Verilog 语言在开发之初就与 C 语言的目的不同，所以二者的可比性不高。本书将从 VHDL 语言的最基本特性开始介绍，我们认为你以前没有学过硬件描述语言，但是具有计算机语言的基本知识，像 C 语言及其编译和执行的基本流程。

每学习一门新的语言，我们就要学习它的字母、词汇、语法、句法规则和语义。学习 VHDL 语言的步骤也基本相同。你也需要学习它的字母、词汇、词义、句法（语法和规则）和语义（语句所代表的含义）。VHDL-87 标准使用 ASCII 字符集，VHDL-93 标准使用 ISO 字符集。ISO 字符集含有 ASCII 字符集和其他重音字符，其中 ASCII 字符集只是 ISO 字符集的前 128 个字符。VHDL 语言包含很多标识符、保留字、特殊符号和变量，均在附录 A 中列出。我们需要理解和掌握 VHDL 语言的语法和句法，并通过一定的规则和结构写出语句，这样我们就可以用 VHDL 语言描述组合逻辑电路和时序逻辑电路了。我们可以通过听说读写来熟练掌握一门交流语言。同样，如果我们想掌握好 VHDL 语言，那么也需要反复地训练，不断地用 VHDL 语言描述各个不同的数字系统。

由于 VHDL 是一种硬件描述语言，所以它同一般的编程语言有所不同。最重要的一点，由于 VHDL 语言是描述硬件的，所以它的有些语句必须并行执行（因为它们所模拟的硬件的各个组成部分是同时操作的）。VHDL 语言广泛地应用于硬件的描述、文档记录、仿真和自动综合，所以它的结构也是为了这些目的量身订造的。在以后的章节中，我们将通过实例介绍用 VHDL 语言设计数字硬件的各种方法。

常用缩写

VHDL：VHSIC 硬件描述语言
VHSIC：超高速集成电路
HDL：硬件描述语言
CAD：计算机辅助设计
EDA：电子设计自动化
LSI：大规模集成
MSI：中规模集成
SSI：小规模集成
VLSI：超大规模集成
ULSI：特大规模集成
ASCII：美国信息交换标准代码
ISO：国际标准化组织
ASIC：专用集成电路
FPGA：现场可编程门阵列
PLA：可编程逻辑阵列
PAL：可编程阵列逻辑
PLD：可编程逻辑器件
CPLD：复杂可编程逻辑器件

2.3　组合逻辑电路的 VHDL 描述

如何表示硬件的并发性操作是一般计算机语言用于模拟硬件要克服的最大困难。通常程序语句是严格按确定的顺序执行的。程序执行时，在每个时刻程序都处于整个流程的一个特定点，按确定的先后顺序执行程序的不同模块。为了模拟组合逻辑电路（内含很多同时工作的逻辑门），我们需要对电路中的不同部分，能够“模拟”它们同时工作（并发性操作）。

VHDL 是通过并发语句来模拟组合逻辑电路的。并发语句总是处于待激发状态，在任何时刻，只要语句右边的信号发生变化，语句就执行，语句左边的信号就得到新的计算值。

下面我们先从一个简单的门电路开始讲解如何用 VHDL 进行描述。如图 2.3 所示的电路，如果每个门均有 5 ns 的传输延迟，那么我们可以用图中的两条语句描述该电路，其中 *A*, *B*, *C*, *D* 和 *E* 表示信号。VHDL 语言中的信号通常与物理系统中的信号直接对应。“<=”是信号赋值运算符，作用是把右边的计算结果赋值给左边信号。上面两个语句进行仿真时，只要 *A* 或 *B* 变化，第一条语句就执行；只要 *C* 或 *D* 变化，第二条语句就执行。设有初始值：$A = 1, B = C = D = E = 0$。若在 0 时刻 *B* 变为 1，则在 5 ns 时 *C* 变为 1，在 10 ns 时 *E* 变为 1。



图 2.3　门电路网络

像上边的例子，VHDL 的信号赋值语句是一个并发语句。VHDL 仿真器监控着每一条并发语句的右边部分，在任何时刻只要信号一改变，就重新执行右边的运算操作，经过一定的延时后，左端的信号重新赋值。这就是时间硬件的工作方式。对于硬件系统，当门电路的输入发生变化时，硬件将重新计算，并且经过门电路的延时后，其输出发生改变。在并发语句中，各个语句的前后顺序位置不同不会影响结果。

一开始，我们可能不考虑电路延时，此时有

```
C <= A and B;
E <= C or D;
```

这表示传输延时为 0 ns。这时，仿真器将添加一个无穷小的延时，记为 Δ。假设开始时 $A = 1, B = C = D = E = 0$。如果 *B* 在 1 ns 时变为 1，则 *C* 将在 1+Δ 时刻发生变化，*E* 将在 1+2Δ 时刻变化。

与顺序执行的程序不同，并发语句的位置顺序不影响结果，例如，

```
E <= C or D;
C <= A and B;
```

仿真的结果会于前面的完全一样。

赋值语句的一般格式如下：

```
信号名 <= 表达式 [after 延时];
```

当语句执行时，表达式就进行操作，左边的信号在仿真进程中排队等候，将在规定的延时后发生改变。中括号里的"**after** 延时"并不是语句的必须部分，它是可选的。如果省略"**after** 延时"，那么信号将在经过Δ延时后更新。注意，语句的执行和信号的更新不是同时进行的。

即使 VHDL 程序中没有明确的循环，并发语句也可以循环反复执行。把一个反相器的输出端连接到输入端上，如图 2.4 所示。若输出为'0'，则该 0 值被反馈到输入端，在经过反相器延时（假设为 10 ns），输出变为'1'。然后这个'1'又反馈回输入端，再经过 10 ns 的传输延迟后，输出变为'0'。信号 *CLK* 将会在'0'和'1'之间持续振荡，其波形如图 2.4 所示。图中对于的 VHDL 并发语句产生同样的结果。如果 *CLK* 开始为 0，那么语句执行并且经 10 ns 延时后，*CLK* 变为 1。因为 *CLK* 发生了变化，使得该语句再次执行，又经过 10 ns 后 *CLK* 变为 0，这个过程将无限循环下去。

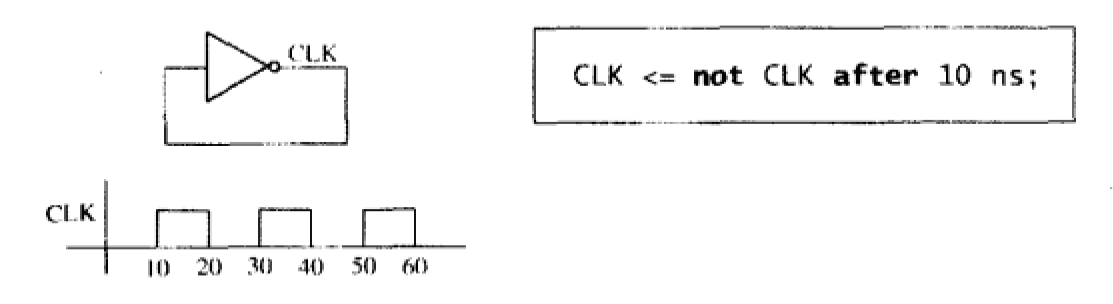


图 2.4　反馈反相器

图 2.4 中的语句可以生成一个周期为 20 ns 的时钟波形。但是，下面的并发语句

```
CLK <= not CLK;
```

却在仿真时将导致运行时间（run-time）错误。由于延时为 0，CLK 的值将在 0 +Δ，0 + 2Δ，0 + 3Δ 等时刻依次变化。由于Δ是无穷小时间，时间都进不到 1ns，只是在原地打转。

一般来说，VHDL 语言对大小写字母不敏感，也就是说，编辑器和仿真器对大写和小写字母做同样的处理。这样，下面的语句：

`Clk <= NOT clk After 10 ns` 和 `CLK <= not CLK after 10 ns` 是等价的。信号名和其他的 VHDL 标识符可以包含字母、数字和下划线(_)。标识符必须以字母开头，而且不能以下划线结尾。因此，C123, ab_23 是合法的标识符，而 1ABC, ABC_是不合法的。每一个 VHDL 语句必须由分号（;）结束。空格、制表（tab）和回车也和字母一样处理。这就意味着，一条 VHDL 语句可以分几行来写，也可以在一行中可以有几条语句。VHDL 代码的所在行里，双画线(--)之后的内容全部视为注释。保留字（也叫做关键字），如 **and, or, after** 等，在 VHDL 语言中有特定的含义，本书中均用黑体表示。

图 2.5 表示有一个公共输入信号 *A* 的三个门电路和相应的 VHDL 代码。当 *A* 发生改变是，三条并发语句同时执行，相当于三个逻辑门同时对信号的改变进行处理。但是，如果每个逻辑门具

有不同的延迟，那么每个逻辑门的输出将在不同时间发生改变。假设三个逻辑门的延迟分别为 2 ns, 1 ns 和 3 ns，若 *A* 在 5 ns 时刻发生改变，则逻辑门输出 *D, E* 和 *F* 分别在 7 ns, 6 ns 和 8 ns 时刻发生改变。图 2.5 中的 VHDL 语句的执行也是如此。即使图中语句是同时执行的，但信号 *D, E* 和 *F* 则将分别在 7 ns, 6 ns 和 8 ns 时刻更新。然而，如果这些逻辑门没有延迟，则 *D, E* 和 *F* 将在 $5+\Delta$ 时刻同时更新。

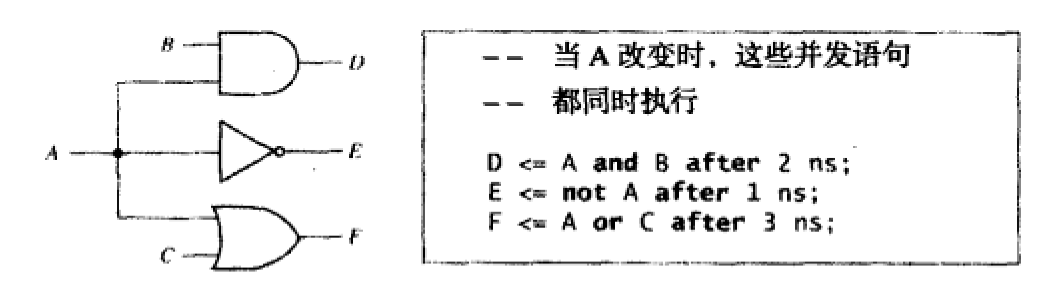


图 2.5 公用一个输入，但具有不同的延迟的三个门

在本例中，每个信号的数据类型都是位（bit），表示它们只能取逻辑‘0’或‘1’（在 VHDL 中，为了与整数数值区别，位的数值用单引号括起来）。

在数字系统设计中，我们经常需要对一组信号进行相同操作。位信号的一维数组定义为**位矢量**（**bit_vector**）。设 *B* 为一个 4 位矢量，其索引为 0 ~ 3，则这个矢量 *B* 的 4 个元素分别为 *B*(0), *B*(1), *B*(2)和 *B*(3)。我们可以用如下语句定义一个位矢量 *B*：

```
B: in bit_vector (3 downto 0);
```

语句 `B <= "1100"` 表示把‘1’赋值给 *B*(3)、‘1’赋值给 *B*(2)、‘0’赋值给 *B*(1)和‘0’赋值给 *B*(0)。

图 2.6 表示一个由 4 个与门组成的阵列。其输入为位矢量 *A* 和 *B*，输出为位矢量 *C*。虽然我们可以用 4 条 VHDL 语句描述这 4 个与门，但是我们也可以直接对位矢量进行与操作，这样更加简便。位矢量的与操作是对矢量的相应位进行与操作。

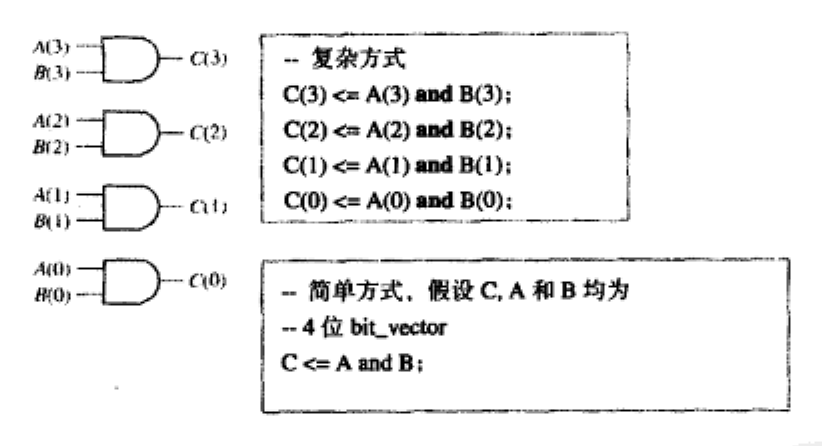


图 2.6 AND 门阵列

2.4 VHDL 模块

一般每个 VHDL 模块都有实体说明和结构体声明。实体说明定义所有的输入信号和输出信号，而结构体说明给出模块的具体内部操作。下面我们举例说明，如图 2.7 所示。实体说明通过 entity 先定义该模块的名字为 two_gates，再通过 port 定义该模块的输入信号和输出信号，A, B 和 D 均为数据类型为位的输入信号，E 为数据类型为位的输出信号。结构体命名为 gates（通过 architecture 引出），信号 *C* 是在构造体内部定义的，说明它是一个内部信号。在关键词 **begin** 和 **end** 之间的两条并发语句描述两个逻辑门的操作。

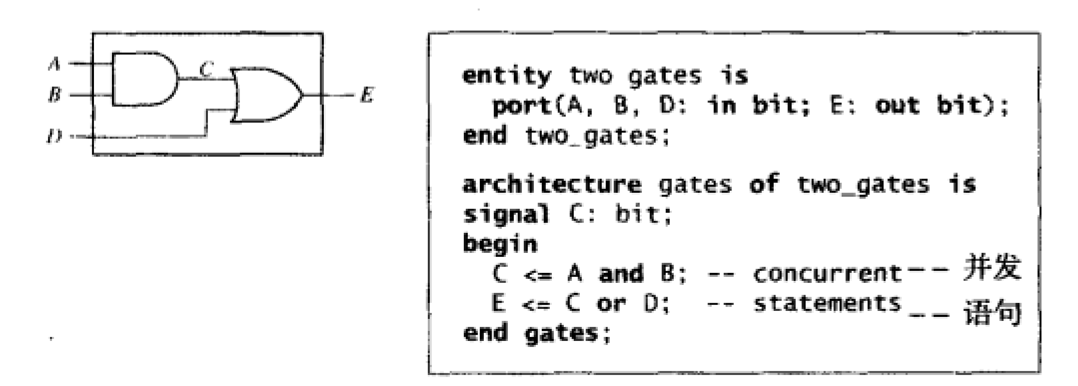


图 2.7　两个逻辑门的 VHDL 模块

如果把我们要设计的模块视为一个黑箱，那么实体部分指出了其与外部的接口（参见图 2.8）。

从前面例举的简单例子中可以看出，我们基于 VHDL 描述一个系统时，在顶层必须首先给出该系统的实体和结构体，同时还要给出该系统中各个模块的实体和结构体（参见图 2.9）。每个实体说明包含了接口信号列表，而这些接口信号用于连接其他模块或者系统外部。实体说明句法为

```
entity 实体名 is
   [port（接口信号说明）;]
end [entity] [实体名];
```

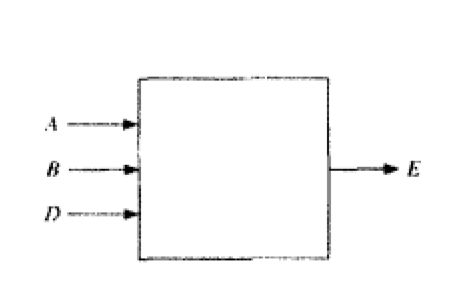


图 2.8　双门模块的黑箱表示

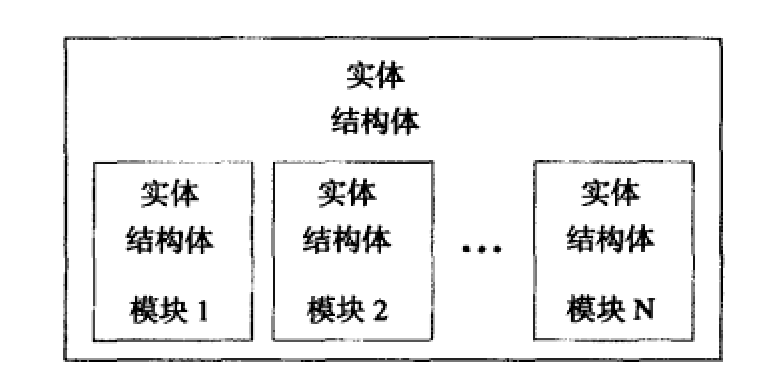


图 2.9　VHDL 程序结构

中括号中的内容可选。接口信号说明的格式为

```
端口信号列表：模式 数据类型名[:=初值]
{; 端口信号列表：端口通信模式 数据类型名[:=初值]};
```

花括号表示没有或者有更多的信号说明语句。**模式**表明端口信号的方向，指出该信号是输入还是输出。输入端口信号的模式为 **in**，输出端口信号的模式为 **out**，输入输出双向端口信号的模式为 **inout**。数据类型确定数据取值的类型或者交换的消息类型。到目前为止，我们所接触到的数据类型只有位（bit）和位矢量（bit-vector），其他的数据类型我们将在 2.10 节中介绍。给信号赋初值是可选的，如果缺省的话，就会赋给一个默认值。例如，

```
port (A, B : in integer : = 2; C, D: out  bit);
```

这里，*A* 和 *B* 是输入信号，数据类型是整数，初值为 2；*C* 和 *D* 是输出信号，数据类型为位，其默认初值为 ‘0’。这些初值只在仿真时起作用，并不影响系统综合。

端口信号模式除了 **in, out** 和 **inout** 外，还有两种模式分别为 **buffer** 和 **linkage**。**buffer** 模式和 **inout** 模式相类似，也可以进行读写。当端口信号只是输出信号时，而我们还想在模块内部对其进行读操作，此时 **buffer** 模式就很有用。当 VHDL 实体要连接在一个非 VHDL 实体时，我们就用 **linkage** 模式。使用这两种模式时会受到一些限制，因此我们最好只使用 **in, out** 和 **inout** 模式。每个实体有一个或者多个结构体与之相联系，一个结构体说明的句法为

```
architecture 结构体名 of 实体名 is
        [定义语句]
begin
        结构体的主体部分
end [architecture] [结构体名];
```

在**定义语句**部分，说明结构体中使用的信号和元件分量（部件）。**结构体的主体部分**包含了描述该模块操作的语句。

下面，我们要写出一个 1 位全加器的实体和结构体。一个 1 位全加器完成两个 1 位二进制数和 1 个进位数的加法并输出一个 1 位和数和一个进位数。如图 2.10 所示，由端口定义可知，*X*, *Y* 和 C_{in} 是位类型输入信号，C_{out} 和 *Sum* 是位类型输出信号。

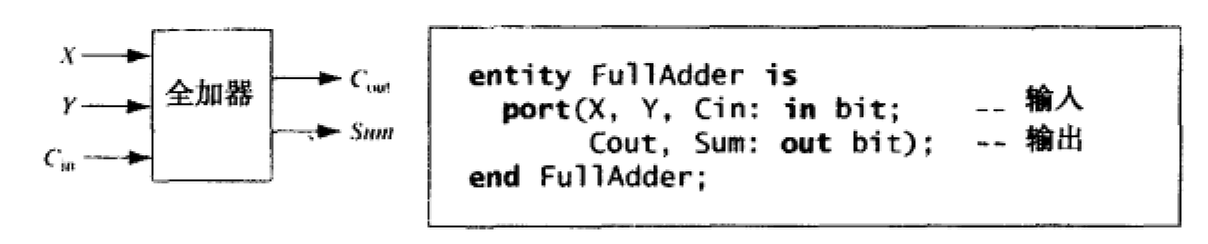


图 2.10　1 位全加器的实体说明

全加器的具体操作由下面的结构体说明给出：

```
architecture Equations of FullAdder is
begin          -- concurrent assignment statements
  Sum  <= X xor Y xor Cin after 10 ns;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

在本例中，结构体名（Equations）是任意的，但是实体名（FullAdder）必须与之前所定义的实体名相同。*Sum* 和 C_{out} 的 VHDL 赋值语句代表了由全加器的逻辑表达式。除此之外，我们还可以用其他的结构体说明方式，如真值表、逻辑门的连接电路等。在 C_{out} 逻辑表达式中，我们用了一些括号，比如（X and Y），这是因为在 VHDL 中没有规定逻辑运算的优先次序，除了 NOT 运算。

2.4.1　四位全加器

下面，利用上边的全加器模块，把它作为一个元件，用 4 个全加器连接构成一个 4 位二进制加法器（参见图 2.11）。首先，我们定义 4 位全加器的实体（参见图 2.12）。由于输入和输出均为 4 位，所以我们定义索引为 3 **downto** 0（我们也可以用索引为 4 to 1 的位矢量）的位矢量。

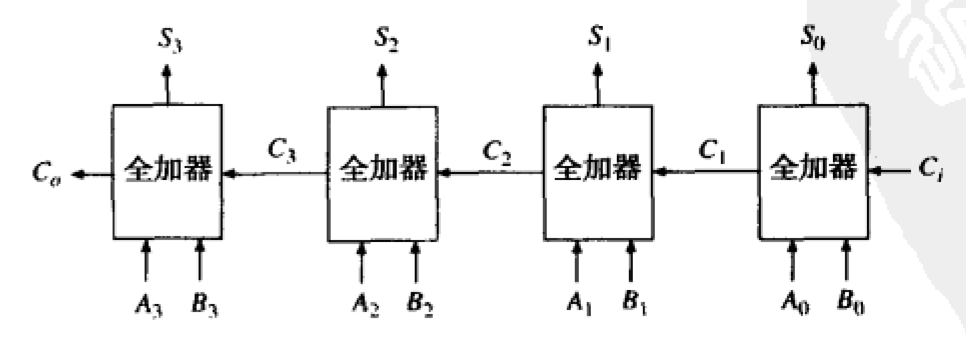


图 2.11　四位二进制加法器

然后，在结构体 Adder4 中把 FullAdder 定义为一个元件（参见图 2.12）。元件说明语句与全加器实体说明语句很类似，并且其输入和输出端口信号名与全加器相对应。不管什么时候，在一

个程序中，如果某一部分生成的模块，在该程序的另一个部分中要使用，则在该程序中要使用元件说明语句。但是，元件说明语句可以不在使用它的程序里，它可以在有原始创建模块的实体和结构体的程序中。为了元件的可重复使用，通常我们会构建元件库，并把元件说明语句放入库文件中。

```
entity Adder4 is
  port(A, B: in bit_vector(3 downto 0); Ci: in bit; -- Inputs
       S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
end Adder4;
architecture Structure of Adder4 is
component FullAdder
  port (X, Y, Cin: in bit;        -- Inputs
        Cout, Sum: out bit);      -- Outputs
end component;
signal C: bit_vector(3 downto 1); -- C is an internal signal
begin     --instantiate four copies of the FullAdder
  FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
  FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
  FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
  FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

图 2.12　用结构描述方法编写的四位加法器程序

元件说明语句后，我们定义一个三位的内部进位信号 C。在结构体的主体部分，我们创建了几个复制的 FullAdder 元件（在 CAD 中，我们使用"instantiate"创建 4 个 FullAdder 的复制）。每一个复制的 FullAdder 都有一个名字（如 FA0）和端口映射表，而且端口映射表的每个信号都与 FullAdder 元件上的端口一一对应。所以 $A(0)$, $B(0)$和 C_i 与输入 X, Y 和 C_{in} 分别对应，$C(1)$和 $S(0)$与输出 C_{out} 和 Sum 分别对应。注意，端口映射表中各个信号的顺序必须与元件说明语句中的各个端口顺序相同。

在仿真前，先将 FullAdder 和 Adder4 的实体和结构体放在同一个文件里进行编译；也可以把 FullAdder 程序提前编译，并把其结果存在一个库中，并在编译 Adder4 程序时调用该库。

本书中所有的仿真实例均用 ModelSimVHDL 仿真器，其他仿真器都用类似的命令文件，并生成类似的输出文件。我们可以用下面的仿真器命令测试 Adder4 程序。

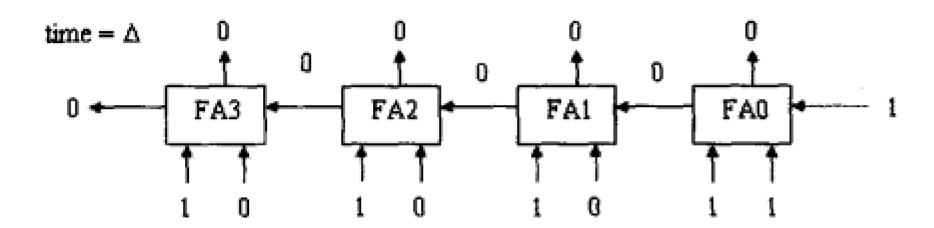
```
Add list A B Co C Ci S    -- 把输入输出信号写入仿真器的显示队列
force A 1111              -- 把输入A 赋为 1111
force B 0001              -- 把输入B 赋为 0001
force Ci 1                -- 把输入Ci赋为 1
run 50 ns                 -- 模拟运行时间为50 ns
force Ci 0
force A 0101
force B 1110
run 50 ns
```

我们之所以把仿真运行时间设为 50 ns，是因为这个时间足够使进位传播经过所有全加器。由上面的仿真命令得到的仿真结果如下：

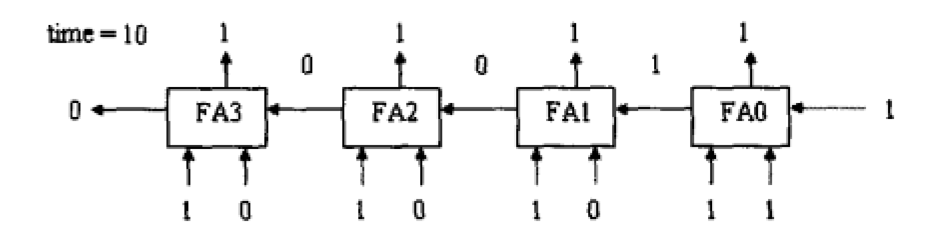
```
ns   delta   a     b     co   c     ci   s
0    +0      0000  0000  0    000   0    0000
0    +1      1111  0001  0    000   1    0000
10   +0      1111  0001  0    001   1    1111
20   +0      1111  0001  0    011   1    1101
```

```
30    +0   1111   0001   0   111   1   1001
40    +0   1111   0001   1   111   1   0001
50    +0   0101   1110   1   111   0   0001
60    +0   0101   1110   1   110   0   0101
70    +0   0101   1110   1   100   0   0111
80    +0   0101   1110   1   100   0   0011
```

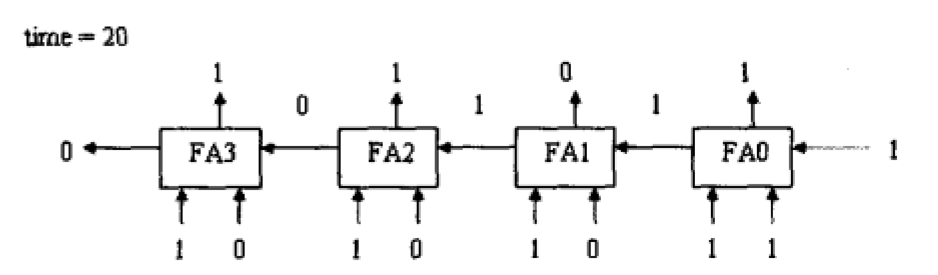
由此我们可以清楚地观察到进位信号是如何传播的。它每 10 ns 向前传播一位。若全加器的输入在Δ时刻发生变化，则有



每一个 FA 计算的和与进位，10 ns 后就出现在 FA 的输出端：



由于 FA1 的输入发生了改变，其输出在 10 ns 后改变：



仿真的最终结果为

1111 + 0001 + 1 = 0001，进位为 1（在 40 ns 时）

0101 + 1110 + 0 = 0011，进位为 1（在 80 ns 时）

该仿真在 80 ns 时刻就停止，因为在那之后每个全加器的输入信号就没有任何变化了。

本节中，我们介绍了如何用实体和结构体创建一个 VHDL 模块。在四位全加器的例子中，我们介绍了如何使用 VHDL 编写结构化的程序。元件应该在结构体之前说明，其句法为

```
component 元件名
   port （端口信号名及其数据类型列表）;
end component;
```

元件说明语句中，**port** 语句与实体说明中 port 语句的结构是一致的。电路中元件的连接是通过元件调用语句来实现的，其格式为

```
标识：元件名 port map （实际信号列表）;
```

实际信号列表中的各个信号要与元件说明中端口信号一一对应。

2.4.2　buffer 模式的使用

考虑图 2.13 给出的例子。设各个变量在 0 ns 时刻均为 0，但是 *A* 在 1 ~ 10 ns 时刻发生改变。

```
entity gates is
   port (A, B, C: in bit; D, E: out bit);
end gates;

architecture example of gates is
begin
  D <= A or B after 5 ns; -- statement 1
  E <= C or D after 5 ns; -- statement 2
end example;
```

图 2.13　不会被编译的 VHDL 程序

图 2.13 的程序代码不会被大多数工具所编译、仿真和综合。这是因为 *D* 的通信模式定义为 out，而语句 2 中却用 *D* 进行赋值运算的缘故，所以 *D* 的模式必须是 **inout** 或 **buffer**（参见图 2.14）。如果使用 **inout** 模式，则在综合工具中生成实际的一个双向信号。实际上，由于 *D* 并不是电路的外部输入，所以这里使用 **buffer** 模式更为恰当。一个 **buffer** 模式用于指出一个信号是电路的外部输出，但是我们可以从实体的结构体内部读出该信号的值。在下面的程序代码中，对于信号 *D*，我们用 **buffer** 模式代替了 **out** 模式。

```
entity gates is
  port(A, B, C: in bit; D: buffer bit; E: out bit);
end gates;

architecture example of gates is
begin
  D <= A or B after 5 ns;   -- statement 1
  E <= C or D after 5 ns;   -- statement 2
end example;
```

图 2.14　使用 buffer 通信模式的 VHDL 程序

一直到 10 ns，所有的信号仍保持为‘0’。当 *A* 在 10 ns 时刻发生改变时，语句 1 就被激活重新计算，在 15 ns 时，*D* 的值变为‘1’，此时语句 2 又被激活重新计算，信号 *E* 在 20 ns 时变为 1。这段代码描述了两个门的具体操作（每个门的延迟均为 5 ns）。

2.5　顺序语句和进程语句

在前几节中介绍的并发语句比较适合模拟组合逻辑电路。组合逻辑电路只对输入的改变做出反应，而同步时序逻辑电路对输入变化的响应则依赖于时钟信号。由于其输出和状态只在有效时钟沿到来时发生改变，所以许多输入的变化就可能被忽略。模拟时序逻辑电路要从模拟基本的选择性有效时钟条件、沿触发的设备、顺序操作等开始。本节中，我们将学习 VHDL 进程语句，它用来模拟时序逻辑。

进程语句的书写格式为

```
process(敏感信号表)
begin
    顺序语句
end process;
```

当执行进程语句时，**begin** 和 **end** 之间的所有语句都是顺序执行的。括号中的变量称为敏感信号表，不管任何时候，只要敏感信号表中的任何信号发生变化时，该进程就会立即执行。例如，如果一个进程语句的起始语句为 **process**（A, B, C），那么当 *A*, *B*, *C* 中的任何一个发生变化时，该进程都会执行。不论何时，只要敏感信号表中的任何一个信号发生改变，进程主体中的顺序语句就按顺序依次执行。当进程执行完之后，进程回到起始语句，等待敏感信号表中的信号再一次发生变化。

如果把并发语句

```
C <= A  and  B ; -- 并发
E <= C  or   D ; -- 语句
```

用于进程语句中，它们就会变为顺序语句，按先后顺序依次执行。因此，我们应该注意，当并发语句在进程外时，它们的先后顺序对结果没有影响，但是一旦它们被放入进程内，那么就会按先后顺序依次执行。

```
process (A, B, C, D)
begin
   C <= A  and  B ; -- 顺序
   E <= C  and  D;  -- 语句
end process;
```

当 *A*, *B*, *C*, *D* 中任何一个发生改变时，进程语句就开始执行。若在进程执行过程中，*C* 发生改变，由于 *C* 是敏感信号，所以该进程将再次执行。

VHDL 进程语句既可以模拟组合逻辑电路，也可以模拟时序逻辑电路。在模拟组合逻辑电路时，我们不是必须要使用进程语句，但是在模拟时序逻辑电路时，我们必须使用进程语句。如果把进程语句用于模拟组合逻辑电路，那么我们应该多加小心。请看图 2.15 给出的程序代码，它用了进程语句。写这一程序代码的初衷是级联两个逻辑门，但是实际上该代码并不表示这种电路。

```
entity nogates is
  port(A, B, C: in bit;
       D: buffer bit;
       E: out bit);
end nogates;

architecture behave of nogates is
begin
  process(A, B, C)
  begin
    D <= A or B after 5 ns;  -- statement 1
    E <= C or D after 5 ns;  -- statement 2
  end process;
end behave;
```

图 2.15　含有进程的 VHDL 程序

该进程的敏感信号表只包含了外部输入 *A*, *B*, *C*。假设在 0 ns 时，所有的信号变量均为 ‘0’。随后，*A* 在 10 ns 时变为 ‘1’，则进程开始执行，进程主体中的语句按顺序执行。由于在执行开始时，*D* 的值没有发生改变，所以语句 2 中 *D* 的值为其初始值。在 15 ns 时，*D* 变为 ‘1’，但是 *E* 仍为 ‘0’。由于 *D* 的值没有传递给 *E*，所以此程序不能描述两个门的级联电路。如果令 *D* 也是敏感信号变量，则进程将再次执行，使 *E* 在 20 ns 时发生改变。虽然这样可以使程序符合要求，但是最好还是用并发语句。

在学习了顺序语句和进程语句之后，我们就可以学习更多的例子。下一节，我们介绍如何用

进程语句模拟简单的触发器，然后再介绍 VHDL 的基本仿真运行过程。在学习了这些之后，我们将展示更多的例子来说明进程的工作过程和仿真的运行过程。

2.6　用进程语句模拟触发器

触发器在输入时钟的上升沿或下降沿发生状态改变。我们可以用 VHDL 语言中的进程语句模拟这一行为。例如，一个简单的 D 触发器在每个时钟 *CLK* 的上升沿改变状态，其输出为 Q。其对应的进程程序代码如图 2.16 所示。

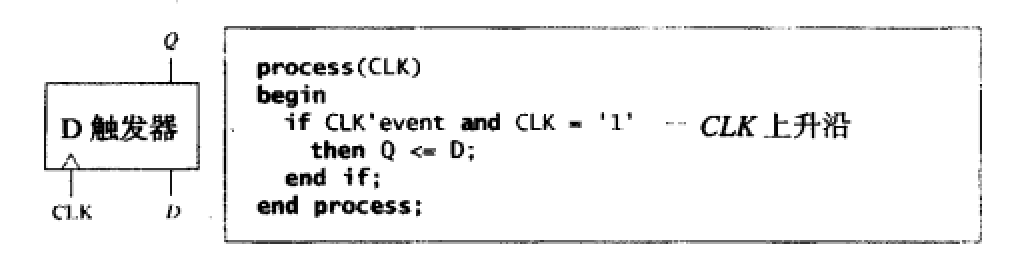


图 2.16　D 触发器的 VHDL 程序

从图 2.16 中可以看出，无论何时，只要 *CLK* 发生变化，则进程就执行一次，并且回到进程的开始等待 *CLK* 的再次改变。其中 **if** 语句用于测试时钟上升沿是否到来，并且在时钟上升沿到来时，Q 等于 D 的值。表达式 `CLK'event` 使设备具有边沿触发的功能。表达式`'event` 为 VHDL 提前定义的信号属性。在 VHDL 中有两种信号属性，它们的返回值分别为数值和信号。表达式`'event` 返回一个数值。当 *CLK* 发生改变时，表达式 `CLK'event` 均为真。如果表达式 *CLK* = '1' 也为真，则意味着时钟从 '0' 变为 '1'，即上升沿。

如果 VHDL 程序只用于仿真目的，则可以用下面的语句

```
if CLK = '1'
    ...
```

来得到对应于时钟上升沿操作的效果。但是，当 VHDL 代码用于硬件综合时，这条语句就会生成锁存器，而表达式 `CLK'event` 就可以生成边沿触发的设备。

如果从时钟上升沿到输出 Q 发生变化，触发器具有 5 ns 的延迟，在上边的进程中，我们可以使用语句 `Q<=D after 5 ns` 代替 `Q<=D`。

在进程中，**begin** 和 **end** 之间的语句是按先后顺序依次执行的。在前面进程中，语句 `Q<=D;` 是一个顺序语句，只在 *CLK* 上升沿到来时执行。相反，并发语句 `Q<=D;` 只要 D 改变就立即执行。如果我们要对上面的进程进行综合，则综合器生成的 Q 必须为触发器，因为只有触发器才能在 *CLK* 上升沿发生改变。如果要对并发语句 `Q<=D;` 进行综合，则综合器将会只在 D 和 Q 之间简单地用一条线或一个缓存器连接。

在图 2.16 中我们应该注意到，由于 D 不会引起触发器状态的改变，所以 D 不在进程的敏感信号表中。图 2.17 给出一个透明锁存器及其 VHDL 程序代码。由于 G = '1'时，D 的改变会引起 Q 的改变，所以 G 和 D 都在进程的敏感信号表中。如果 G 变为'0'，则进程也开始执行，但是 Q 的值不变。

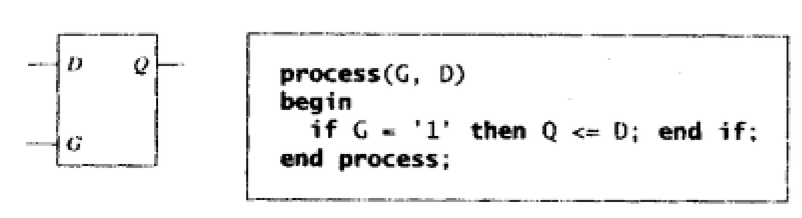


图 2.17　透明锁存器

如果触发器含有一个低电平有效的异步清零输入（*ClrN*），它可以独立于时钟把触发器清零，则我们应该把图 2.16 中的代码进行修改，在进程的敏感信号表中加入 *ClrN*，得到带有异步清零端的 D 触发器的 VHDL 程序（参见图 2.18）。这样，无论 *CLK* 和 *ClrN* 之中任何一个发生变化，进程都会执行。由于异步信号 *ClrN* 的优先级比 *CLK* 高，所以先对 *ClrN* 进行检测。如果 *ClrN* 为'0'，则触发器清零。否则，对 *CLK* 进行检测，当时钟上升沿到来时，*Q* 值更新。

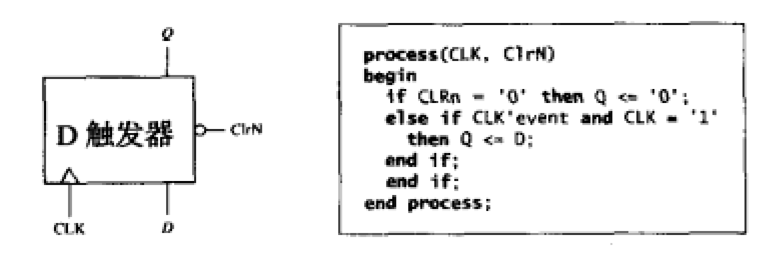


图 2.18　具有异步清零端的 D 触发器的 VHDL 程序

在前面的例子中，我们使用了两种顺序语句：信号赋值语句和 **if** 语句。**if** 语句的基本格式为：

```
if   条件   then
  顺序语句1
else   顺序语句2
end if;
```

其中条件为布尔表达式，即它的值只能为“真”或“假”。如果为“真”，则顺序语句 1 执行；如果为“假”，则顺序语句 2 执行。

在 VHDL 语言中，作为顺序语句，**if** 语句可以在进程中使用，但是不能在进程外作为并发语句使用。

if 语句的一般格式为

```
if      条件 then
        顺序语句
{ elsif   条件   then    顺序语句}
 --可以包含0个或多个elsif语句
[ else   顺序语句]
end if;
```

其中大括号表示可以包含任意个数的 **elsif** 语句，中括号表示 **else** 语句可以省略。图 2.19 表示如何用 **if** 嵌套语句描述流程图。其中 C1, C2, C3 表示条件（真或假），S1, S2 ,···, S8 表示顺序语句。每一个 **if** 都需要有相应的 **end if** 结尾，但是 **elsif** 不需要。

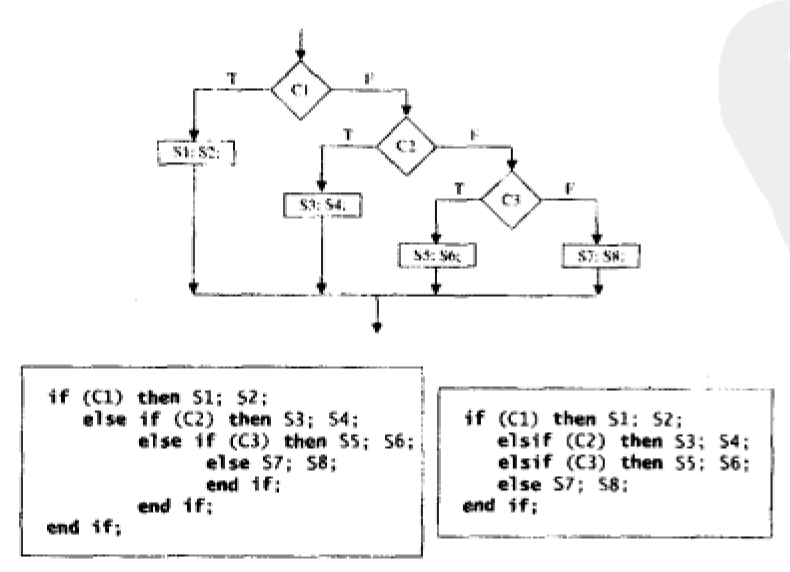


图 2.19　**if** 嵌套语句描述流程图

下面我们来编写一个 J-K 触发器（参见图 2.20）的 VHDL 程序模块。该 J-K 触发器有两个低电平有效的异步输入端 *SN* 和 *RN*，而且在时钟下降沿改变状态。本章中，我们用后缀 N 表示低电平有效(负逻辑)信号。为了方便，本例中我们假设 *SN* = *RN* = 0 的情况不会出现。

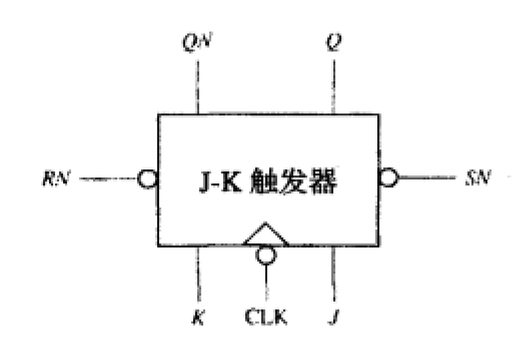


图 2.20　J-K 触发器

该 J-K 触发器 VHDL 程序代码参见图 2.21。在实体的端口说明中定义了输入信号和输出信号。在结构体中，我们定义了 Q_{int} 信号，此信号表示触发器的内部状态。在 **begin** 之后的两条并发语句把此内部状态传输到触发器的输出 *Q* 和 *QN*。我们之所以这样做，是因为端口输出信号不能在结构体内赋值语句的右边出现。这也是解决图 2.13 中存在问题的另一种方法。触发器对 *SN*, *RN* 和 *CLK* 的变化做出响应，所以这三个信号都在进程的敏感信号表中。由于 *RN* 和 *SN* 可以独立于时钟，对触发器进行复位和置位，所以它们将首先被检测。当 *RN* 和 *SN* 均为‘1’时，我们再检测时钟下降沿是否到来。只有在 *CLK* 刚刚从‘1’变为‘0’时，条件（`CLK'event and CLK = '0'`）才为真。J-K 触发器的下一状态特征方程为

$$Q^+ = JQ' + K'Q$$

```
entity JKFF is
  port(SN, RN, J, K, CLK: in bit;    --输入
       Q, QN: out bit);
end JKFF;

architecture JKFF1 of JKFF is
signal Qint: bit;                        -- Qint 既可以作为输入也可以作为输出
begin
  Q <= Qint;                             -- 把 Q and QN 输出到端口
  QN <= not Qint;                        -- 组合输出
                                         -- 外部进程
  process(SN, RN, CLK)
  begin
    if RN = '0' then  Qint <= '0' after 8 ns;      -- RN='0' 时触发器复位
    elsif SN = '0' then Qint <= '1' after 8 ns;    -- SN='0' 时触发器置位
    elsif CLK'event and CLK = '0' then             -- CLK 下降沿
      Qint <= (J and not Qint) or (not K and Qint) after 10 ns;
    end if;
  end process;
end JKFF1;
```

图 2.21　J-K 触发器模型

8 ns 的延迟表示当 *SN* 或 *RN* 变为‘0’时时置位或复位触发器的输出所需的时间。10 ns 的延迟表示时钟下降沿到来后多长时间 *Q* 值改变所需的时间。

2.7 含有 Wait 语句的进程

进程的另一种形式是使用 wait 语句，而不用敏感信号表。但是一个进程中不可以既有敏感信号表又有 wait 语句。含有 wait 语句的进程的格式为

```
process
begin
     顺序语句
     wait语句
     顺序语句
     wait语句
     ...
end process;
```

这个进程会执行顺序语句直到遇到一个 **wait** 语句，它会一直等到 **wait** 语句中的条件得到满足。然后它会执行下一组顺序语句直到遇到下一个 **wait** 语句。这种方式会一直下去直到进程结束。然后，它会再次回到进程的开始。

wait 语句有三种不同的格式：

```
wait  on  敏感信号表;
wait  for  时间表达式;
wait  until  布尔表达式;
```

wait 语句的第一种格式一直等到敏感信号表中的一个信号改变。例如，**wait on** A, B, C；语句一直等到 *A*, *B* 和 *C* 中至少一个发生变化时，进程才继续进行。第二种格式的 **wait** 语句，则等到时间表达式得到满足时，进程才能继续。如果语句为 **wait for** 5 ns，则进程继续执行之前要等待 5 ns。如果语句为 **wait for** 0 ns，则进程等待 Δ 时间。语句 **wait for** xxx ns 在进行 VHDL 程序仿真时非常有用，但是在将要进行综合的 VHDL 程序代码中最好不要使用，因为它们不能被综合。对于第三种的 **wait** 语句，每当布尔表达式中任何一个变量发生变化时，都会进行布尔表达式的计算，当表达式的值为真时，进程将继续执行。例如，

```
wait  until  A = B;
```

会等待直到 *A* 或 *B* 发生变化。然后，判断 $A = B$ 的真伪，若真则进程继续执行 wait 后面的顺序语句。否则，进程将继续等待，直到 *A* 或 *B* 再次发生变化，并且 $A = B$ 为真。

一个进程中不可以既有敏感信号，又有 wait 语句，不可以把一些信号放到敏感信号表中，而其他一些变量则放到 wait 语句中。

当一个 VHDL 仿真器初始化后，每个带敏感信号表的进程会先执行一遍，然后回到进程的起始处等待直到敏感信号表中的信号发生变化。如果一个进程中有 wait 语句，它先执行直到遇到一个 **wait** 语句时。下面两个进程是等价的：

```
process (A, B, C, D)
begin
   C<= A and B after 5 ns;
   E<= C or B after 5 ns;
end process;
```

```
process
begin
   C<= A and B after 5 ns;
   E<= C or B after 5 ns;
   wait on A, C, C, D;
end process;
```

在进程末尾的 **wait** 语句可以代替进程开始的敏感信号表。因此，这两个进程均可以初始化执行顺序语句直到 *A*, *B*, *C* 或 *D* 发生变化。

在一个进程中的顺序语句的执行顺序不必与其中的信号更新的顺序一致。考虑下面的例子：

```
process
begin
  wait until clk'event and clk = '1';
  A<= E after 10 ns;        -- (1)
  B<= F after 5 ns;         -- (2)
  C<= G;                    -- (3)
  D<= H after 5 ns;         -- (4)
end process;
```

这个进程等待的是时钟上升沿。假设时钟上升沿出现在 20 ns 时刻。语句(1), (2), (3), (4)立刻按顺序执行。*A* 的值应该在 30 ns 时变为 *E* 的值；*B* 的值应该在 25 ns 时变为 *F* 的值；*C* 的值应该在 20 + Δ 时变为 *G* 的值；*D* 的值应该在 25 ns 时变为 *H* 的值。随着仿真时间的进展，首先 *G* 发生变化，接着 *F* 和 *D* 在 25 ns 时刻发生变化，最后在 30 ns 时刻 *E* 发生变化。当 *clk* 变为'0'时，**wait** 语句将重新计算判断条件，但它会一直等待到 *clk* 变为'1'，然后继续再执行其他剩余的语句。

如果一个进程中的多个语句在某一个时间同时更新一个信号，则最后一个值为信号更新值。例如，

```
process (CLK)
begin
   if  CLK'event  and  CLK = '0'  then
     Q <= A; Q <= B; Q <= C;
   end if;
end process;
```

每当 *CLK* 从'1'变为'0'时，在经过 Δ 时间后，*Q* 将变为 *C*。

一个进程中必须有敏感信号表或有 wait 语句。图 2.22 所示的 VHDL 程序是不能进行仿真的，因为它既没有敏感信号表，也没有 wait 语句。

```
entity gates is
  port (A, B, C: in bit; D, E: out bit);
end gates;

architecture exam of gates is
begin
  process
  begin
     D <= A or B after 5 ns; -- statement 1
    E <= C or D after 5 ns; -- statement 2
  end process;
end example;
```

图 2.22　不会被仿真的 VHDL 程序

本节中，我们既介绍了带有敏感信号表的进程，又介绍了带有 wait 语句的进程。一个进程中的语句称为顺序语句，因为它们都是按先后顺序执行的。相对地，并发语句只有在右边信号发生变化时，语句才开始执行。信号赋值语句既可以作为并发语句，也可以作为顺序语句，但是 if 语句永远是顺序语句。

2.8 两种 VHDL 延迟：传输延迟和惯性延迟

如果在本章开篇中介绍的例子中使用下列语句描述一个具有 5 ns 传输延迟的 AND 门(与门):

```
C <= A and B after 5 ns;
```

此语句可以描述 AND 门的延迟，但是仍旧引入了一定的复杂度，很多读者不希望这样。如果你要仿真的该 AND 门的输入，相对于门延迟（比如 1ns, 2ns, 3ns 等）、变化较快，则仿真输出将不会反映这种变化。这是由 VHDL 延迟的工作机理造成的。

VHDL 提供了两种延迟——传输延迟和惯性延迟。惯性延迟是 VHDL 的默认延迟方式，像前面的 **after** 语句就表示惯性延迟。惯性延迟概念与读者的一般理解有偏差。

惯性延迟用于模拟逻辑门和其他设备不能把窄脉冲从输入传输到输出。如果一个逻辑门具有理想的惯性延迟 *T*，则它除了把输入信号延迟 *T* 时间以外，还对任何宽度小于 *T* 的脉冲都进行拦截。例如，如果一个逻辑门的惯性延迟为 5 ns，则宽度为 5 ns 的脉冲可以通过，而宽度为 4.999 ns 的脉冲将被拦截通不过。实际设备并不是这样工作的，它对一些宽度很窄的脉冲进行拦截，但是所有宽度小于延迟范围的脉冲不一定都通不过。VHDL 语言就可以模拟这种设备，只拦截宽度很窄的脉冲。我们可以在赋值语句中，通过添加一个 **reject** 语句，模拟这种拦截宽度小于惯性延迟时间的脉冲行为。该语句格式为

```
信号名<= reject 脉冲宽度 after 延迟时间
```

执行该语句时，任何宽度小于惯性延迟时间的脉冲均会被拦截，过一段延迟时间后，信号赋予该语句执行的结果。注意，在此类语句中脉冲宽度必须小于延迟时间。

VHDL 语言中的另一种延迟为传输延迟。它用来模拟由连线引起的延迟，它只是把输入信号延迟指定的一段时间。为了模拟这种延迟，我们在代码中使用 **transport** 语句。图 2.23 说明了传输延迟和惯性延迟的区别。考虑下面的 VHDL 语句：

```
Z1 <= transport X after 10 ns;    -- transport delay
Z2 <= X after 10 ns;              -- inertial delay
Z3 <= reject 4 ns X after 10 ns;  -- delay with specified rejection pulse width
```

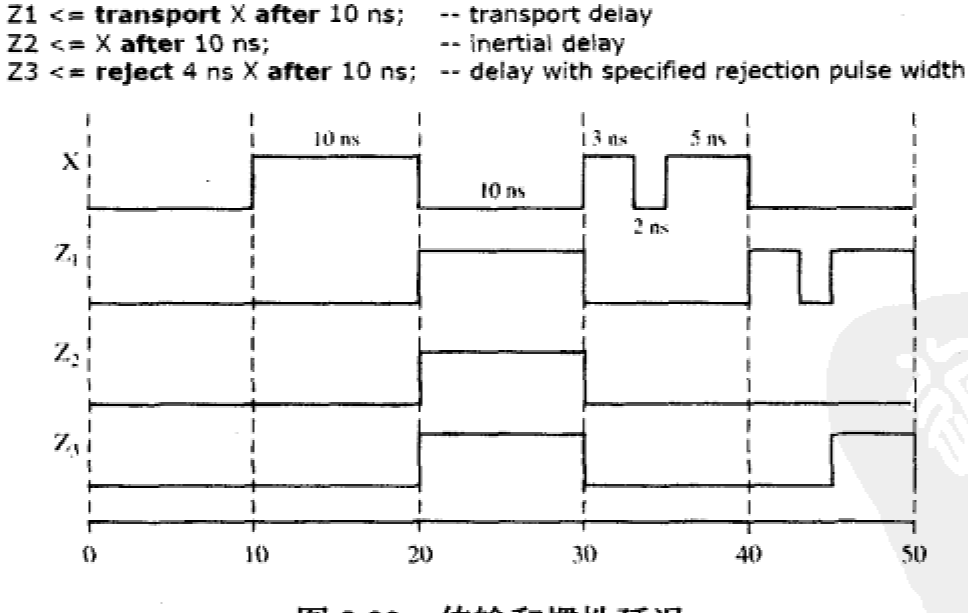


图 2.23 传输和惯性延迟

Z_1 为 *X* 延时 10 ns 后的波形。Z_2 与 Z_1 类似，只是 Z_2 过滤掉了 *X* 中短于 10 ns 的脉冲。Z_3 与 Z_2 相同，只是 Z_3 只滤掉 Z_1 中宽度小于 4 ns 的脉冲。

通常，一个 **reject** 语句与联合使用惯性延迟和传输延迟是等效的。这里给出的 Z_3 语句可以用并发语句代替：

```
Zm<= X after 4 ns; ----惯性延时拒绝短脉冲
Z3<= transport Zm after 6 ns;-----总延时为10 ns
```

注意，这些延迟的引入仅仅是为了仿真。如果理解了惯性延迟的工作机理，则在初次进行 VHDL 仿真时可以少走很多弯路。惯性延迟对脉冲的拦截性，可以用来禁止输出的多变。在用基本逻辑门和简单电路进行仿真时，必须保证所用的测试序列的脉宽比待测模块的惯性延迟要宽。

2.9　VHDL 代码的编译、仿真与综合

一个数字系统完成 VHDL 设计之后，对其 VHDL 代码的仿真是很重要的，理由有两个：第一、需要验证 VHDL 代码是否正确地实现了设计功能要求；第二、需要验证是否满足设计指标要求。我们先对该设计进行仿真，然后针对目标技术（如 FPGA 或定制 ASIC）对其进行综合。本节中，我们先介绍仿真的基本步骤，然后再介绍综合。如图 2.24 所示，VHDL 代码的仿真有三个阶段：分析（编译）、细化和仿真。

对一个数字系统的 VHDL 设计进行仿真之前，必须先对 VHDL 代码进行编译。VHDL 编译器（也称为分析器）首先检查 VHDL 源代码是否符合 VHDL 语句的语法和语义规则。如果存在一个语法错误（如缺少一个分号）或语义错误（如将类型不匹配的两个信号相加），那么编译器将会输出一个错误信息。此外，编译器还检查库文件的引用是否正确。如果 VHDL 源代码符合所有的规则，则编译器就会生成可用于仿真器和和综合器的中间代码。

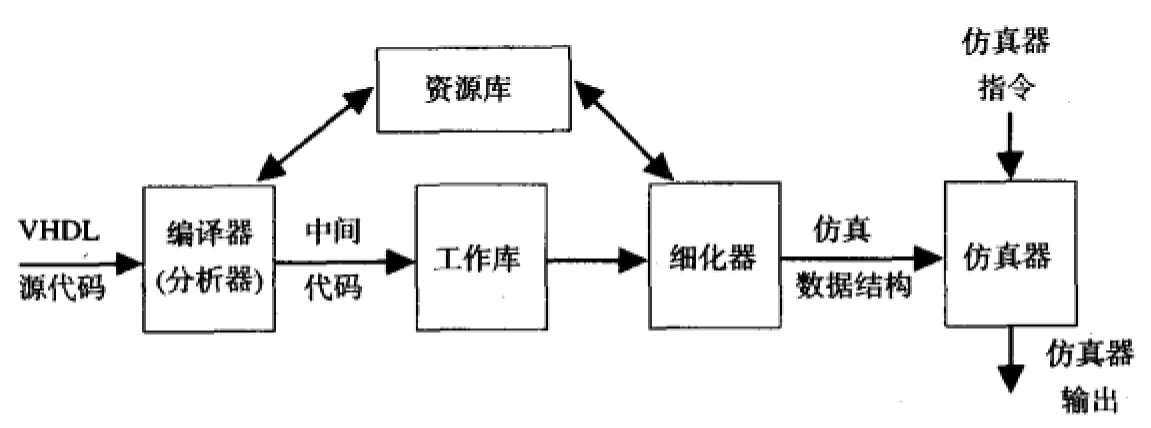


图 2.24　VHDL 代码的编译、细化和仿真

为了进行最后的仿真，VHDL 的中间代码还必须转化为一种可被仿真器直接使用的形式，这一过程称为**细化**（elaboration）。细化过程中，为每个信号创建一个驱动器。每个驱动器保存信号的目前值和将来取值的序列，每当一个信号要在将来的某个时刻发生改变时，该时间和更新值均会被放入该序列中。同时，细化为每一个元件创建端口，为所需的信号分配存储单元，确定这些端口信号的连接关系，并且建立 VHDL 语句执行的前后顺序。最后，细化生成代表该数字系统的数据结构用于仿真。

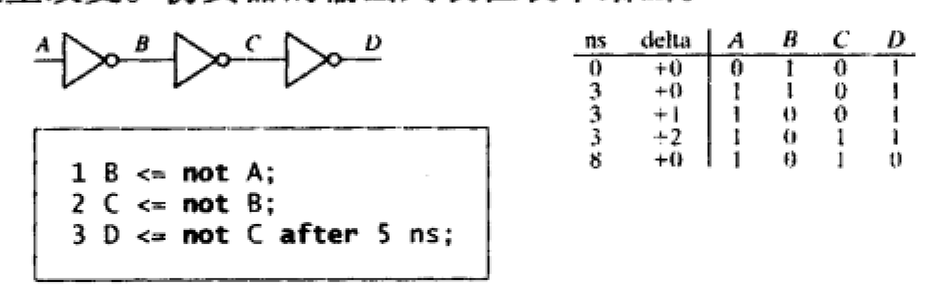
整个仿真过程由**初始化阶段**和实际仿真阶段组成。仿真器接收仿真命令，这些命令用于控制数字系统的仿真，并指定所需的仿真器输出。VHDL 仿真其实是**离散事件仿真**。在这种仿真中时间的进程是离散的。初始化阶段用于给信号赋初值。在仿真过程中，VHDL 语句执行后，给相应的操作安排调度。这些操作称为事务（transaction），而这个过程称为**事务调度**(scheduling a transaction)。当语句执行时，安排调度的操作不必此时发生，只有在调度时间发生即可，事务并不意味着信号有取值的变化。事务处理后，信号的新值可能跟旧值相同。如果信号的取值发生了变化，我们就说一个**事件**（event）发生。

为了正确地进行初始化，则在 VHDL 程序模块中必须对初始值进行设定。如果没有任何初始

值的设定，则一些仿真器就根据信号的数据类型对其进行初始设定。必须注意，初始化只是为了仿真，而不是为了综合。初始化过程的仿真时间设置为 0，且每个进程都被激活。进程开始“执行”，并进行相应的事务调度。但是，不到调度时间的到来，调度的事务是不会发生的。进程执行了一次后，就等待敏感信号表中的信号发生变化。

理解延迟时间 Δ 的作用，对于解释仿真器的输出很重要。虽然仿真器的输出波形没有反映出 Δ 延迟，但是它们反映在输出列表中。仿真器使用 Δ 延迟以保证信号按照恰当的顺序进行计算。仿真器的基本操作过程如下：当一个元件的输入改变时，给输出取值安排一个调度，在给定的延迟后发生改变，如果没有给定延迟时间，则输出在 Δ 时间后发生改变。仿真器处理完所有输入信号的变化后，仿真时间向前推进一步，在处理确定所有输出信号的变化。如果时间前进量为有限值（例如 1 ns），则 Δ 计数器复位，仿真重新开始。除非处理完与当前仿真相关的所有 Δ 延迟，否则仿真时间是不会向前推进的。

下面的例子说明图 2.25 中的电路是如何进行仿真的。假设 A 在 3 ns 时刻发生改变。语句 1 执行，给 B 安排一个调度，在 3 + Δ 时刻发生改变。接着，时间前进到 3 + Δ 时刻，语句 2 执行，给 C 安排一个调度，在 3 + 2Δ 时刻发生改变。时间前进到 3 + 2Δ 时刻，此时语句 3 执行，给 D 安排一个调度，在 8 ns 时刻发生改变。也许你会认为 D 应该在(3 + 2Δ + 5) ns 时刻发生改变，但是由于时间前进量为一个有限值，所以 Δ 计数器复位。因此，如果某一事件的调度时间是将来某一有限时间，则可以忽略 Δ 延迟。由于在 8 ns 后没有任何调度安排，所以仿真器进入空闲模式，等待输入信号再次发生改变。仿真器的输出列表在表中给出。




```
1 B <= not A;
2 C <= not B;
3 D <= not C after 5 ns;
```

ns	delta	A	B	C	D
0	+0	0	1	0	1
3	+0	1	1	0	1
3	+1	1	0	0	1
3	+2	1	0	1	1
8	+0	1	0	1	0

图 2.25 并发语句仿真时的 delta 延迟

2.9.1 多进程仿真

如果一个模块中有两个或两个以上的进程，则所有的进程都与其他进程一起并发执行的。如果在进程外有并发语句，则这些语句也是一起并发执行的。每个进程中的语句都是顺序执行的。一个进程的执行不需要时间，除非它内有 **wait** 语句（例如 wait for 10 ns、wait for 0 ns 和 wait on E）。如果没有给定延迟，则信号在 Δ 时间后进行更新。

作为多进程仿真的一个例子，我们跟踪图 2.26 给出的 VHDL 代码的执行。关键词 **transport** 定义一个传输延时。

```
entity simulation_example is
end simulation_example;

architecture test1 of simulation_example is
signal A,B: bit;
begin
  P1: process(B)
  begin
    A <= '1';
    A <= transport '0' after 5 ns;
  end process P1;

  P2: process(A)
  begin
    if A = '1' then B <= not B after 10 ns; end if;
  end process P2;
end test1;
```

图 2.26 进程仿真的 VHDL 代码

图 2.27 给出了信号 A 和 B 的驱动器在仿真过程中的工作情况。细化过程结束后，驱动器存有 ‘0’，这是位（bit）数据类型的默认初始值。仿真开始后，先进行初始化。两个进程同时执行一次，并等待敏感信号表中信号的变化。在 0 时刻，进程 P_1 开始执行，给 A 安排两个变化的调度（在 time = Δ 时刻，A = ‘1’；在 time = 5 ns 时刻，A = ‘0’）。在 0 时刻，进程 P_2 也同时开始执行，但是由于 A 在 time = 0 时刻始终为‘0’，所以 B 没有发生改变。仿真时间向前推进到 time = Δ 时刻，A 变为‘1’，A 的改变使进程 P_2 再次执行。由于 A = ‘1’，给 B 安排调度，在 time = 10 ns 时 B = ‘1’。下一个变化的调度时间为 time = 5 ns，此时信号 A 的值变为‘0’。这一变化又引起进程 P_2 的执行，但由于 A = ‘0’，所以 B 不变仍为‘0’。B 的值将在 time = 10 ns 时刻变为‘1’。这一时刻的 B 的变化，又引起进程 P_1 开始执行，给 A 又安排了两个变化的调度。在 time = 10 + Δ 时刻，A 变为‘1’，进程 P_2 又开始执行，给 B 安排一个调度，在 time = 20 ns 时刻发生改变。之后，在 time = 15 ns 时 A 变为‘0’，仿真器如此反复进行仿真，一直到运行时间结束为止。我们应该注意理解 A 是在 time = 15 ns 时发生改变，而不是在 time = 15 + Δ 时发生改变。只有当没有设定延迟时，Δ 延迟才起作用。

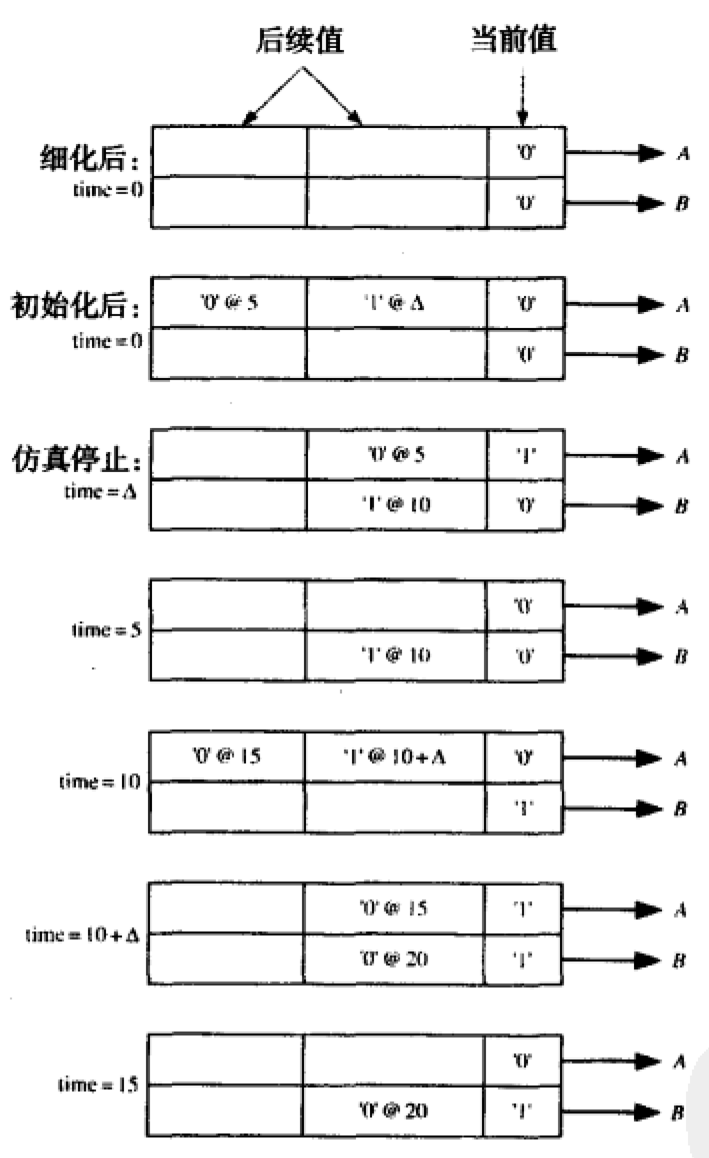


图 2.27 仿真过程中信号驱动器的工作情况

VHDL 的开发之初是为了仿真，但现在 VHDL 最重要的应用就是综合或者基于 VHDL 描述自动生成硬件。VHDL 综合软件可以按照要求把 VHDL 程序翻译成由所需的元件构成的电路。VHDL 无论用于仿真还是综合，其初始步骤（分析和细化）都是相同的，如图 2.24 所示。仿真和综合步骤如图 2.28 所示。

综合可以与仿真并行进行，但是一般在仿真之后进行，这是因为设计者希望能在综合前找到错误所在。如果一个数字系统的 VHDL 程序已经进行了仿真并且被验证可以正常工作，则该 VHDL 程序就可以进行综合，生成所需的元件和它们之间的互连列表。综合器的输出可以用于实现具体

硬件（例如 CPLD、FPGA 或 ASIC）。用于综合的 CAD 软件可以生成 CPLD 或 FPGA 硬件编程所需的信息。在 ASIC 实现中，CAD 软件还可以生成 ASIC 所需的掩模。基于 VHDL 的数字逻辑的实现和综合，在以后的章节中进行更详细讨论。

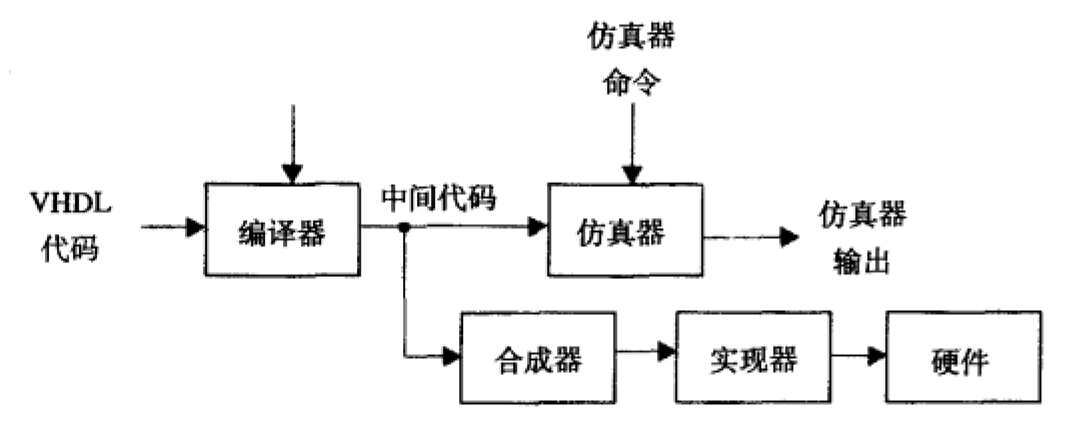


图 2.28　VHDL 程序的编译、仿真和综合

2.10　VHDL 数据类型和运算符

2.10.1 数据类型

VHDL 语言中有多个预定义的数据类型。信号既可以采用预定义的数据类型，也可以使用用户自定义的数据类型。下面对一些预定义的数据类型加以简单说明：

位（bit）	逻辑 0 或 1
布尔量	逻辑假或真
整数	整数，取值范围为 $-(2^{31}-1) \sim +(2^{31}-1)$（在某些实现中取值范围更宽）
实数	浮点数，取值范围为–1.0E38 ~ +1.0E38
字符	包含上划线、小写字母、数字和特殊符号在内的任何合法 VHDL 字符（每个可显字符必须用单引号括起来，如‘d’, ‘7’, ‘+’）
时间	带单位的整数，单位为 fs, ps, ns, us, ms, sec, min, h

注意，整数在 VHDL 语言中的取值范围只能是 32 位二进制数，即 $-(2^{31}) \sim +(2^{31}-1)$。

枚举类型（**enumeration** type）是一种很普遍的用户定义数据类型。使用枚举类型时，要把所有可能的值都例举出来。例如，

```
type state_type is (S0, S1, S2, S3, S4, S5);
signal state : state_type : = S1;
```

上边的语句定义了一个初始值为 S_1 的信号 *state*，它可以取 $S_0, S_1, S_2, S_3, S_4, S_5$ 中的任何一个值。如果没有给出初始值，那么默认初始值为枚举值最左端的值，本例中为 S_0。

由于 VHDL 语言对数据类型要求严格，所以一般来说不同数据类型的变量和信号不可以同时出现在同一条赋值语句中，而且 VHDL 语言没有数据类型自动转换。所以语句

```
A<= B or C;
```

只有在 *A, B, C* 为同一种数据类型或是十分相关的数据类型时才有效。如果数据类型不匹配，则必须进行数据类型转换或生成“重载运算符”。IEEE 包集合中提供的重载运算符将在 2.13 节中进行介绍。

2.10.2 VHDL 语言的运算符

VHDL 语言的预定义运算符可以分为 7 类：

1. 二进制逻辑运算符: **and or nand nor xor xnor**
2. 关系运算符: = / = < <= > >=
3. 移位运算符: **sll srl sla sra rol ror**
4. 加法运算符: + – &(拼接)
5. 正负运算符: + –
6. 乘法运算符: *** / mod rem**
7. 辅助运算符: **not abs ****

不用括号时，第 7 类的优先级最高，最先进行运算，接着是第 6 类、第 5 类，依次类推，第 1 类的优先级最低，即最后进行计算。在同一类中运算符的优先级都相同，在表达式中按“从左到右”的顺序依次计算。添加括号可以改变运算顺序。观察下面的表达式，其中 *A*, *B*, *C* 和 *D* 均为位矢量（bit_vector）:

```
(A & not B or C ror 2 and D) = "110010"
```

注意这是一个关系运算表达式，用以检验等号是否成立，它并不是赋值语句。

计算该表达式时，运算符的操作顺序为

```
not, &, ror, or, and, =
```

设 *A* = "110", *B* = "111", *C* = "011000", *D* = "111011"，则该运算的具体过程为

```
1. not B = "000" (按位取反)
2. A & not B = "110000" (拼接)
3. C ror 2 = "000110" (循环右移 2 位)
4. (A & not B) or (C ror 2) = "110110 (逻辑或)
5. (A & not B or C ror 2) and D = "110010" (逻辑与)
6. [(A & not B or C ror 2 and D) = "110010"] = TRUE (加括号进行验证，所得结果正确)
```

二进制逻辑运算符（第 1 类）及 **not** 运算符可以用于数据类型为位、布尔量、位矢量和布尔矢量的计算，但要求两个操作数的数据类型是相同的，所得结果的数据类型也与操作数一致。

关系运算符(第 2 类)，运算所得结果均为布尔量——真(TURE)或假(FALSE)。“=”和“/ = ”可以用于任何数据类型的计算，而其他关系运算符只能用于数值、枚举和一些数组类型的运算中。例如，若 *A* = 5, *B* = 4, *C* = 3，则表达式(A<= B) **and** (B<= C)的结果为假（FALSE)。

移位运算符可以用于任何位矢量和布尔矢量的运算。下面的例子中，A 为位矢量且等于 "10010101"时，则

```
A sll 2 得 "01010100"    (逻辑左移，用 '0'填补空位)
A srl 3 得"00010010"     (逻辑右移，用 '0'填补空位)
A sla 3 得"10101111"     (算术左移，用最左端位填补空位)
A sra 2 得"11100101"     (算术右移，用最右端位填补空位)
```

```
A rol 3 得"10101100"        （循环左移）
A ror 5 得"10101100"        （循环右移）
```

加法运算符+，–可以应用于整数和实数运算，这种运算在 bit 或 bit-vector 数据类型下还没有定义。这就是我们为什么在全加器中，对每个位特别设定 *sum* 和 *carry* 位（参见图 2.12）。但是，现在有很多标准库中允许位矢量（bit-vector）的加法操作。如果使用这些库，我们就可以直接用语句 *C*<= *A* + *B* 实现加法操作即可。我们将在 2.13 节中介绍常用库。

&（拼接符）可以把两个矢量拼接在一起形成一个大的矢量。例如，"010"&'1'得"0101"；"ABC"&"DEF"得"ABCDEF"。

*和/运算符可以用于整数或浮点操作数的乘除运算。mod 和 rem 运算符用于计算整数的模和取余运算。**运算符用于整数和浮点数的乘幂运算。abs 用于计算数值的绝对值。

2.11 简单综合示例

综合工具通过“查看”VHDL 代码来试推所需的硬件元件。为了正确地综合代码，必须要遵守一些规则：当编写 VHDL 代码时，必须时刻牢记你是在设计硬件，而不是简单地编写计算机程序。每一个 VHDL 语句都蕴涵某种硬件需求。因此，不好的 VHDL 代码会导致差的硬件设计。即使 VHDL 代码在仿真时给出正确的结果，综合后生成的硬件系统不一定正常工作。时序问题会导致硬件工作不正常，即使仿真的结果是正确的。

请看图 2.29 的 VHDL 代码（注意 *B* 不在进程的敏感信号表中）。该代码仿真过程为：无论何时只要 *A* 发生变化，则进程就执行一次。在进程开始执行后，*C* 的值将反映在进程开始时的 *A* 和 *B* 的值。如果 *B* 发生变化，则不会引起进程的执行。

```
entity Q1 is
  port(A, B: in bit;
       C: out bit);
end Q1;

architecture circuit of Q1 is
begin
  process(A)
  begin
    C <= A or B after 5 ns;
  end process;
end circuit;
```

图 2.29 仿真和综合导致不同输出时的 VHDL 代码举例

如果综合该程序代码，大多数综合器都会生成如图 2.30 所示的或门电路。综合器将会发出警告提示 *B* 不在敏感信号表中，但是综合继续进行直到生成合适的电路。综合器也将忽略上面语句中的 5 ns 延迟。如果你想要模拟恰好 5 ns 的延迟，则应该使用计数器。由于当 *B* 发生改变时，进程并不执行，所以仿真器的输出与综合器的输出将不匹配。这说明综合器进行综合时，并不是完全按照你的意思去做，它认为你很可能想要一个或门，所以就生成一个或门电路（伴随着警告）。但是，该电路的功能却与综合前仿真不一样。因此，我们应该好好检查综合器给出的敏感信号表中的缺信号的警告，这一点很重要。这样综合器可以帮你，或许它生成了你不想要的硬件。

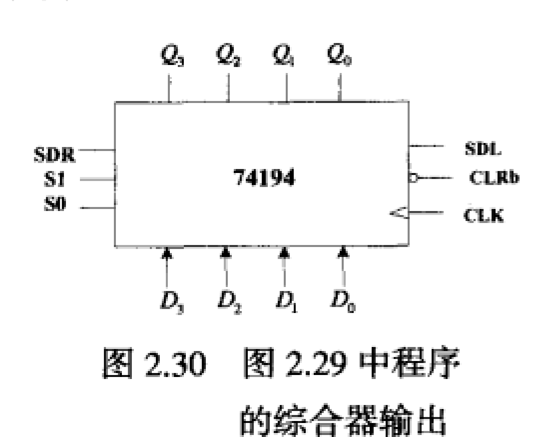


图 2.30 图 2.29 中程序的综合器输出

请看图 2.31 给出的 VHDL 代码。如果要对该代码进行综合，你会得到什么样的硬件呢？

```
entity Q3 is
  port(A,B,F, CLK: in bit;
      G: out bit);
end Q3;

architecture circuit of Q3 is
signal C: bit;
begin
  process(Clk)
  begin
    if (Clk = '1' and Clk'event) then
      C <= A and B;  -- 语句1
      G <= C or F;   -- 语句2
    end if;
  end process;
end circuit;
```

图 2.31　VHDL 程序实例

下面让我们思考该代码所表示的电路的框图，我们先不考虑该电路内部的具体结构。电路的框图如图 2.32 所示。隐藏具体细节的抽象能力是优秀系统设计的重要组成部分。

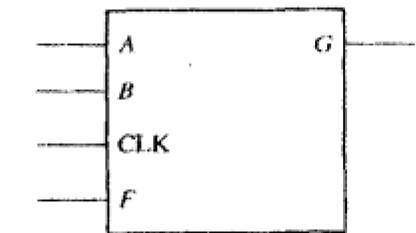


图 2.32　图 2.31 中 VHDL 程序的框图

注意到 C 为内部信号，因此它在框图中没有出现。

现在让我们考虑框图的内部结构。该电路不是由两个级联的门构成的，信号的赋值语句在进程中。信号赋值语句之前的时钟语句中使用 `clk'event`，表示该电路是由时钟的有效沿触发的。因为有效时钟沿过后，C 和 G 的值都需要保持，所以 C 和 G 都需要触发器。请注意，在该进程中，在语句 1 中更新的 C 值不会影响语句 2 的执行，它只在下一次进程中才会起作用，为此使用触发器保存 C 的值。所以该代码所蕴涵的硬件如图 2.33 所示。

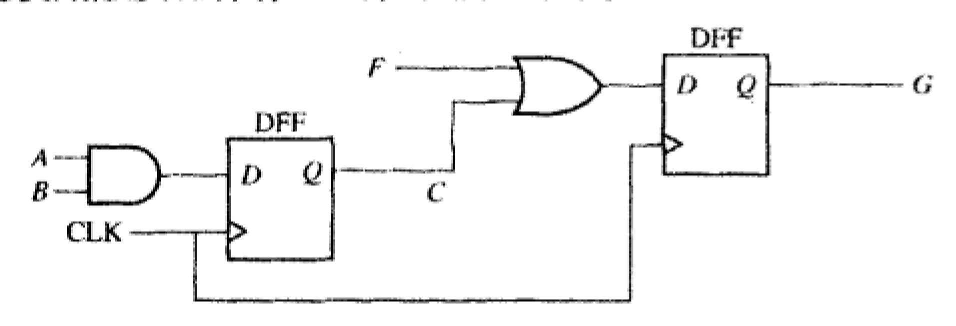


图 2.33　图 2.31 中 VHDL 程序对应的硬件

我们注意到下面的代码就表示一个 D 锁存器：

```
process (G, D)
begin
   if G = '1' then Q<= D; end if ;
end process;
```

下面我们讨论一下为何该代码不能表示一个输入为 G 和 D 的与门电路。如果 G = '1'，则一个与门的输出与 `if` 语句的结果相当。但是，如果当前 Q = '1'，且 G 变为'0'，那么会怎么样呢？当 G 变为'0'时，一个与门把它直接传输到输出（输出为 0），而我们这里模拟的这个电路却不是这样。因为当 G 不等于'1'时，该电路的输出不应改变。因此，该电路模块是一个 D 锁存器，而不是一个与门。

如果想用触发器或寄存器来实现，而它们在时钟上升沿改变状态，则我们使用下面的 **if** 语句即可：

```
if clock 'event and clock = '1' then ... end if;
```

大多数综合器都支持该语句。在上边的 **then** 和 **end if** 之间的赋值语句来说，赋值语句左边的信号在综合时都会生成一个触发器或寄存器。这个例子的寓意是：如果你不想生成不必要的触发器，那么不要把信号赋值语句放进时钟进程中。如果 clock'event 被省略，则综合器将生成锁存器，而不是触发器。

现在，考虑图 2.34 给出的 VHDL 代码。如果你要对该代码进行综合，则综合器将会生成一个空框图。这是因为上面框图中的输出 *D* 从未被赋值。综合器会给出如下警告提示：

```
Input <CLK> is never used
Input <A> is never used
Input <B> is never used
Output <D> is neverassigned
```

```
entity no_syn is
  port(A,B, CLK: in bit;
       D: out bit);
end no_syn;

architecture no_synthesis of no_syn is
  signal C: bit;
begin
  process(Clk)
  begin
    if (Clk='1' and Clk'event) then
      C <= A and B;
    end if;
  end process;
end no_synthesis;
```

图 2.34 不能进行综合的 VHDL 程序

2.12 多路选择器的 VHDL 设计

多路选择器是一种组合逻辑电路，它可以只用并发语句实现，也可以用进程实现。条件信号赋值语句，例如 **when** 或使用 **with select** 的选择信号赋值语句，都可以用来外描述多路选择器，而不用进程。进程中的 case 语句也可以用来模拟多路选择器。

2.12.1 并发语句的使用

图 2.35 表示一个具有两个数据输入端和一个控制输入端的 2 选 1 多路选择器如（MUX），输出为 $F = A' \cdot I_0 + A \cdot I_1$，相应的 VHDL 语句为

```
F <= (not A and I0) or (A and I1);
```

这里使用了一个并发信号赋值语句就描述了该多路选择器。同样，我们也可以用条件信号赋值来表示该多路选择器，如图 2.35 所示。只要 A, I_0 或 I_1 发生改变时，则该语句就执行。当 A = '0' 时，MUX 的输出为 I_0；否则输出为 I_1。在这个条件语句中，I_0, I_1 和 F 可以是位（bit）或位矢量（bit-vector）数据类型。

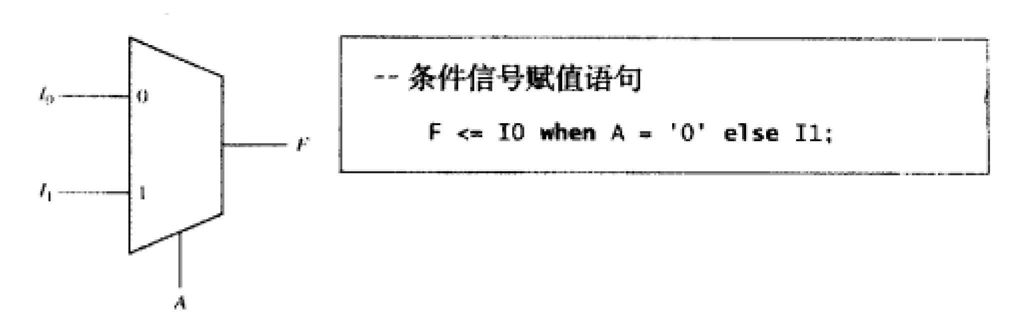


图 2.35　2 选 1 多路选择器

条件信号赋值语句的一般格式为

```
信号名 <= 表达式1  when  条件1
      else 表达式2  when  条件2
      [else  表达式N];
```

当表达式或条件中的一个信号发生改变时，该并行语句就执行。如果条件 1 为真，则信号的值等于表达式 1 的值。否则，如果条件 2 为真，则信号的值等于表达式 2 的值，依此类推。中括号里的内容为可选项。图 2.36 说明了两个级联的 MUX 怎样用一个条件信号赋值语句来表示。当 E = '1'时，第 2 个 MUX 输出选择 A；否则，它选择第 1 个 MUX 的输出（当 D = '1'时为 B；否则为 C）。

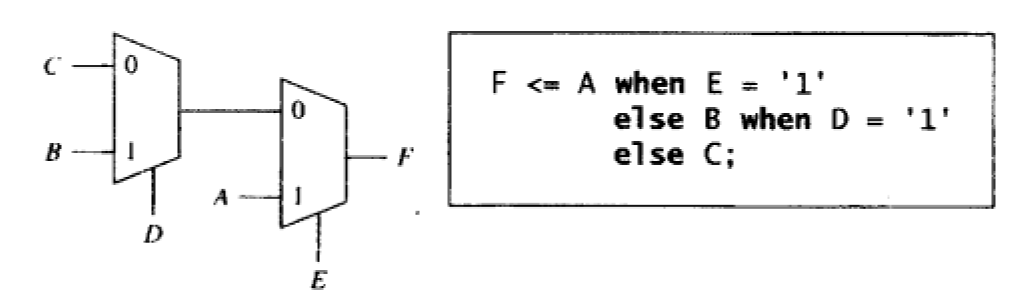


图 2.36　级联 2 选 1 多路选择器

一个 4 选 1 多路选择器（MUX）有四个数据输入端和两个控制输入端 A 和 B。控制端的输入决定四个输入数据中哪个数据传送到输出端的端口号。4 选 1 多路选择器的逻辑表达式为：

$$F = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$$

该表达式对应的 VHDL 语句为

```
F <= (not A and not B and I0) or (not A and B and I1) or
     (A and not B and I2) or (A and B and I3);
```

我们还可以用条件赋值语句模拟该 4 选 1 多路选择器：

```
F <= I0  when  A & B = "00"
else  I1  when  A & B = "01"
else  I2  when  A & B = "10"
else  I3;
```

表达式 A & B 表示将 A 与 B 拼接，即把 A 和 B 拼接起来生成一个 2 位的矢量。通过检查该位矢量，选择相应的 MUX 输入。例如，如果 A = '1', B = '0'，则 A & B = "10"，且选择 I_2。如果不使用拼接运算符，则我们可以使用更复杂的条件：

```
F <= I0  when  A = '0'  and  B = '0'
else  I1  when  A = '0'  and  B = '1'
else  I2  when  A = '1'  and  B = '0'
else  I3;
```

4 选 1MUX 的第 3 种 VHDL 模型，可以用选择信号赋值语句，如图 2.37 所示。A & B 不能直接用于这种类型语句当中，所以我们可以先把 A & B 的值赋给 sel，再用 sel 的值选择 MUX 的输入并赋值给 F。

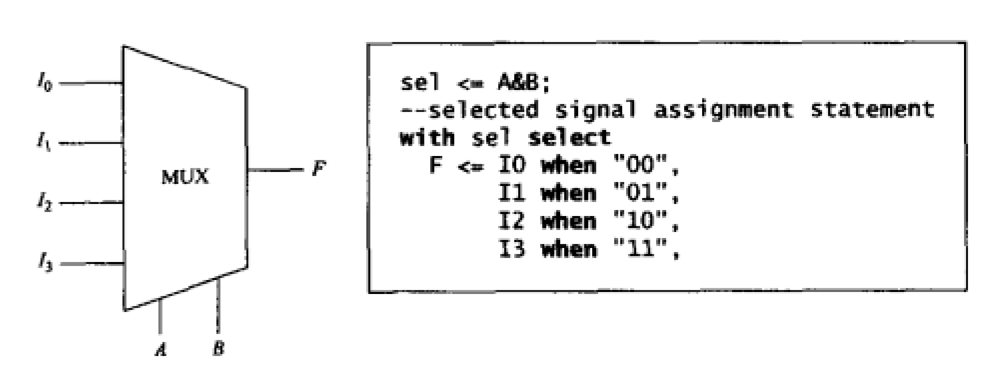


图 2.37　4 选 1 多路选择器

选择信号赋值语句的一般格式为

```
with  表达式-s  select
   信号-s  <=表达式1  [after 延迟时间]  when 条件1,
              表达式2  [after 延迟时间]  when 条件2,
                ...
              [表达式n  [after 延迟时间]  when others];
```

只要任何一个表达式中的一个信号发生变化，该并发语句就执行。首先计算表达式-s。如果表达式等于条件 1，则将表达式 1 赋给信号-s；如果等于条件 2，则将表达式 2 赋给信号-s；等等。如果该语句列出了表达式-s 的所有可能取值，则应去掉最后一条语句，否则保留最后一条语句。当有这一行语句时，如果表达式-s 不等于任何一个列出的条件时，将表达式 n 赋给信号-s。经过给定的延迟或者在 Δ 时间（如果省略了 **after** 延迟时间）后，就会更新信号。

2.12.2　进程的使用

如果在一个进程中使用一个 MUX 模块，则就不能使用并发语句。作为另一个方法，我们可以用一个 **case** 语句来描述 MUX：

```
case Sel is
   when 0 = > F <= I0;
   when 1 = > F <= I1;
   when 2 = > F <= I2;
   when 3 = > F <= I3;
end case;
```

case 语句的一般格式为

```
case 表达式 is
   when 条件1  => 顺序语句1
   when 条件2  => 顺序语句2
   . . .
   [when others  => 顺序语句]
end case;
```



该语句执行时先判断表达式的值，如果表达式的值等于条件 1，那么执行顺序语句 1；如果表达式的值等于条件 2，那么执行顺序语句 2，依次类推。表达式所有的可能值都要列出来，否则必须有 **when others** 语句。

我们已经注意到组合电路的描述既可以使用并发语句，也可以使用顺序语句；而时序电路的描述，则一般需要进程语句。但进程语句在时序电路和组合电路中均可以使用。

2.13 VHDL 语言的库

VHDL 库和包集合通过定义数据类型、函数、元件和重载运算符，扩展了 VHDL 的功能进行扩展。在标准 VHDL 中，有些操作运算只针对特定的数据类型有效。如果其他数据类型也需要使用这些操作运算，则必须使用函数"重载"构建一个"重载"运算符。"函数重载"这个概念在很多通用编程语言中都用。它是指两个或两个以上的函数可以具有相同的名字，只要变量的数据类型差别比较大足以区分实际想要的函数就可以。我们也可以构造重载函数来完成不同类型数据的处理运算。

在 CAD 刚出现时，每个工具生产商都制作各自的库和包集合。在这种情况下，从一个环境到另一个环境的设计转移就成为了一个问题。IEEE 标准库和包集合的出现使设计转移变得容易。最初的 VHDL 标准里只定义了 2 值逻辑（bit 和 bit-vector）。最早的扩展就是把定义扩展到多值逻辑，并作为 IEEE 标准。包集合 IEEE.std_logic_1164 定义了一个 9 值的 std_logic 类型，包括'0'，'1'，'X'（未知）和'Z'（高阻）。此包集合还定义了 std_logic_vector，即为 std_logic 类型的矢量形式。该标准为 std_logic 和 std_logic_vector 定义了逻辑操作和其他功能函数，但是没有提供任何算术操作。std_logic_1164 包集合及其在仿真和综合中的应用将在第 8 章进行详细介绍。

随着 VHDL 更广泛地应用于综合，IEEE 又推出了两个包集合用于编写可综合代码，它们是 IEEE.numeric_bit 和 IEEE.numeric_std。前一个使用位矢量（bit_vector）表示无符号和有符号二进制数，后一个则使用 std_logic_vector。这两个包集合都定义了针对无符号和有符号数的重载逻辑和算术运算符。在第 8 章中，我们将使用 numeric_bit 包集合和无符号数实现算术操作。

要想访问库中的函数和元件，你必须使用库语句和 use 语句。下面的语句

```
library IEEE;
```

允许你的设计访问 IEEE 库中的所有包集合。语句

```
use IEEE.numeric_bit.all;
```

允许你的设计使用整个 numeric_bit 包集合，此包集合包含在 IEEE 库中。无论何时，只要一个包集合被用于一个模块中，库语句和 use 语句就必须放置在该模块实体说明之前。

numeric_bit 包集合定义无符号和有符号类型为不受限制的位（bit）数组：

```
type unsigned is array (natural range <>) of bit;
type  signed  is array (natural range <>) of bit;
```

有符号数用二进制补码形式表示。该包集合中还包含针对无符号和有符号数的算术、关系、逻辑和移位操作的各种重载运算符。

无符号和有符号数类型通常为 bit-vector。但是，重载运算符是针对这两个类型定义的，而不是针对 bit-vector 本身。语句

```
C<= A + B;
```

当 *A*, *B* 和 *C* 都为 bit-vector 时，该语句会引起编译错误。如果这些信号为无符号或有符号类型，则编译器将会调用恰当的重载运算符并进行加法操作。

numeric_bit 和 numeric_std 包集合定义了下面的重载算子：

一元：abs，—
运算：+，—，*，/，rem，mod
关系：=，/ ，=>，<，>=，<=
逻辑：not，and，or，nand，nor，xor，xnor
移位：shifr_left，shift_right，rotate_left，rotate_right，sll, srl, rol, ror

一元运算需要一个单独的有符号操作数。算术、关系和逻辑运算符需要左右各一个操作数。对于算术和关系运算符，下面的一对左右操作数是可以接受的：有符号数和有符号数、有符号数和整数、整数和有符号数、无符号数和无符号数、无符号数和自然数、自然数和无符号数。对于逻辑算子，左右操作数必须都是有符号数或者都是无符号数。当+或–被用于计算不同长度的无符号操作数时，长度较短的操作数通过在左边补 0 的方法进行扩展。丢弃进位，则结果同较长的操作数位数相同。例如，操作下面的无符号数

"1011" + "110" = "1011" + "0110" = "0001"

时，舍去进位。numeric_bit 包集合提供了把一个整数和一个无符号数做加法运算的重载运算符。但是，不能求一个 bit 和无符号类型的和。这样，如果 *A* 和 *B* 均为无符号数，则允许 **A + B + 1**。但是下面的语句

```
Sum <= A + B + carry;
```

中若 carry 是 bit 类型，则该语句不允许使用。因此，为了把 carry 加到无符号矢量 **A + B** 之前，必须将其转化为无符号类型。我们可以用关键词 **unsigned'(0 =>carry)** 来完成这一转换。

图 2.38 给出了一个行为描述的 VHDL 代码，它使用 numeric_bit 包集合中的重载运算符描述一个带进位输入的 4 位加法器。该程序的实体声明部分与图 2.12 的基本一样，只是用 unsigned 类型代替了 bit_vector 类型。由于两个 4 位二进制数相加得到的和为 5 位二进制数，所以在结构体内定义了一个 5 位信号（Sum5）。如果我们计算 *A* + *B*，则计算结果只有 4 位。由于我们想要得到 5 位的结果，所以我们必须把 *A* 扩展成 5 位（在 *A* 的左端添加 0），而 *B* 将会自动扩展来匹配。在使用 numeric_bit 包集合中的重载运算符计算出 Sum5 后，我们把它分成一个 4 位的和（*S*）与一个 1 位的进位（C_o）。大多数综合工具对图 2.38 中的程序代码进行综合后，结果均为一个带有进位输入和输出的加法器。某一版本的 Xilinx 综合器的结果如图 2.39 所示。

```
library IEEE;
use IEEE.numeric_bit.all;

entity Adder4 is
  port(A, B: in unsigned(3 downto 0); Ci: in bit;  -- Inputs
       S: out unsigned(3 downto 0); Co: out bit);  -- Outputs
end Adder4;

architecture overload of Adder4 is
signal Sum5: unsigned(4 downto 0);
begin
  Sum5 <= '0' & A + B + unsigned'(0=>Ci);  -- adder
  S <= Sum5(3 downto 0);
  Co <= Sum5(4);
end overload;
```

图 2.38　无符号矢量 4 位加法器的 VHDL 程序

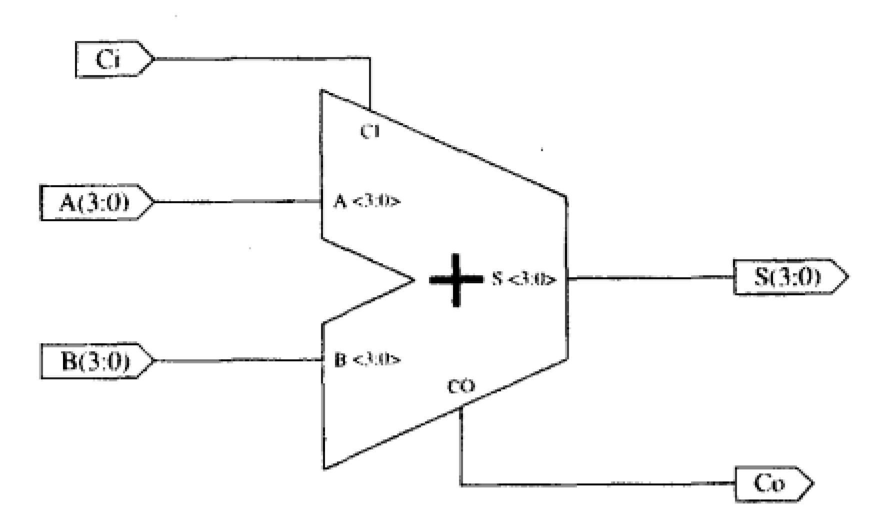


图 2.39　图 2.38 中 VHDL 代码的综合器输出

numeric_bit 包集合中包含的常用转换函数为

```
TO_INTEGER(A):将无符号矢量数A转换成一个整数
TO_UNSIGNED(B, N):将整数转换成长度为N的无符号矢量
UNSIGNED(A): 使编译器把位矢量A作为一个无符号矢量进行处理
BIT_VECTOR(B): 使编译器把一个无符号矢量B作为一个位矢量进行处理
```

如果需要使用多值逻辑，则我们必须使用 IEEE 标准 numeric_std 包集合，而不是 numeric_bit 包集合。numeric_std 包集合定义的无符号和有符号类型为 std_logic 矢量，而不是 bit_vector。为了使用此包集合，我们需要使用三条语句：

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
```

与 numeric_bit 包集合一样，该包集合也定义了一系列针对无符号和有符号数的重载运算符和函数。

另外，一个常用于多值逻辑仿真和综合的 VHDL 包集合为 std_logic_arith 包集合，它是由 Synopsis 公司开发的。该包集合定义的重载运算符同 IEEE numeric_std 包集合很类似，但是它定义的转换函数具有不同的名字，同时还有一些其他的不同之处。std_logic_arith 包集合的最主要的缺点就是它没有对无符号和有符号矢量的逻辑操作进行定义。该包集合不是 IEEE 标准集合，即使它通常放置在 IEEE 的库中。

还有另外一个选择，就是使用 std_logic_unsigned 包集合，它也是由 Synopsis 公司开发的。此包集合没有定义无符号类型，但是定义了一些针对 std_logic_vector 的重载算术运算符。这些运算符把 std_logic_vector 当做无符号数进行操作，当该包集合与 std_logic_1164 包集合联合使用时，就可以对 std_logic_vector 进行算术和逻辑操作，这是因为 1164 包集合定义了逻辑操作。std_logic_unsigned 包集合不是 IEEE 标准集合，即使它通常放置在 IEEE 的库中。图 2.40 中使用 std_logic_unsigned 包集合对如图 2.38 所示 4 位加法器的 VHDL 代码进行了重新编写。由于此包集合提供的重载运算符可以把 std_logic bit 与 std_logic_vector 做加法运算，所以操作前无需进行类型转换。该代码的综合结果与图 2.38 相同。

本节中，我们讨论了四种不同的包集合，它们分别提供了各种用于算术和关系操作的重载运算符。我们最初使用 numeric_bit 包集合，因为它是 IEEE 标准中最简单、最容易使用的包集合。从第 8 章开始，我们将使用 IEEE numeric_std 包集合。这是因为它是一个提供多值信号，且功能与 numeric_bit 包集合相似的 IEEE 标准包集合。我们将不使用 std_logic_arith 和 std_logic_unsigned 包集合。因为它们都不是 IEEE 标准包集合，而且它们的功能没有 IEEE

numeric_std 包集合全面。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity Adder4_v3 is
  port(A, B: in std_logic_vector(3 downto 0); Ci: in std_logic; --输入
      S : out std_logic_vector(3 downto 0); Co: out std_logic); --输出
end Adder4_v3;

architecture overload of Adder4_v3 is
signal Sum5: std_logic_vector(4 downto 0);
begin
  Sum5 <= '0' & A + B + Ci; --adder
  S <= Sum5(3 downto 0);
  Co <= Sum5(4);
end overload;
```

图 2.40 使用 std_logic_unsigned 包集合的 4 位全加器的 VHDL 程序

2.14 用 VHDL 进程语句模拟寄存器和计数器

当几个触发器在同一个时钟沿改变时，表示这些触发器的语句可以放置在同一个时钟进程中。图 2.41 给出了三个触发器构成的循环移位寄存器。这三个触发器均在时钟上升沿都会改变状态。我们假设从时钟沿出现到输出改变之间存在 5 ns 的传输延迟。紧跟着在时钟沿的到来，进程中的三个触发器立即没有延迟地按顺序执行。给新的 Q 值安排一个调度，将在 5 ns 后发生改变。如果我们忽略延迟，并用下列语句代替顺序语句，则得到的结果基本相同。

```
Q1<= Q3;   Q2<= Q1;   Q3<= Q2;
```

在开始的 0 时刻，三条语句无延迟地按顺序执行，然后经过 Δ 延迟后 Q 值改变。在这两种情况下，Q_1, Q_2 和 Q_3 的旧值均用于计算新值。起初，这看起来有点儿奇怪，但是硬件就是这么工作的。在时钟上升沿到来时，所有的 D 输入都被置入触发器，但是经过一段传输延迟后状态才发生改变。

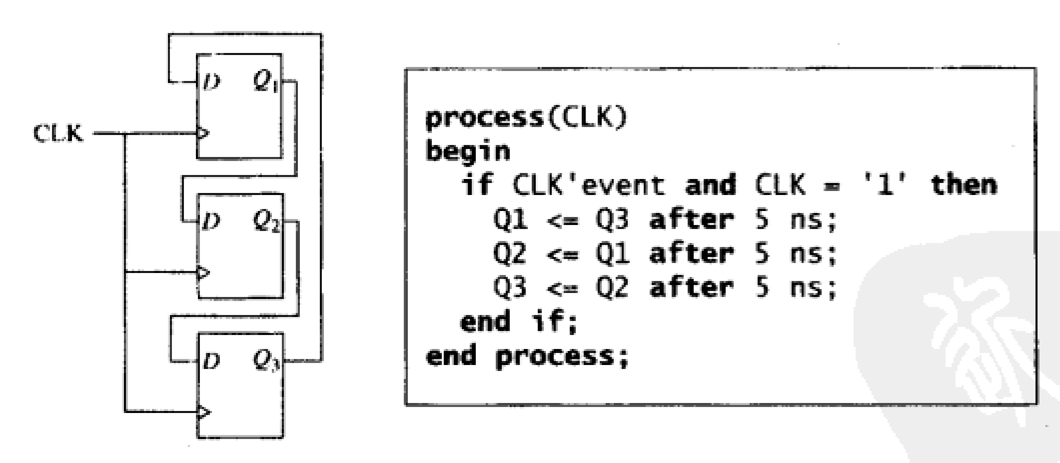


图 2.41 循环移位寄存器

图 2.42 是一个简单的寄存器，它在时钟上升沿置数或清零。如果 CLR = ‘1’，则寄存器清零；如果 Ld = ‘1’，则 D 输入的值被置入寄存器中。这个寄存器与时钟是完全同步的，使得输出 Q 只随着时钟沿的到来而改变，而不随着 Ld 或 CLR 的改变而改变。对此寄存器的 VHDL 编码中，Q 和 D 是范围从 3 到 0 的位矢量（bit-vector）。由于寄存器的输出只在时钟上升沿发生改变，所以 CLR 不在敏感信号表中，它是在时钟上升沿到来后才被检测。如果 CLR = Ld = ‘0’，则 Q 的值不

发生改变。由于 *CLR* 在 *Ld* 之前进行检测，如果 *CLR* = '1'，则 **elsif** 语句会阻止 *Ld* 被检测，*CLR* 优先于 *Ld*。

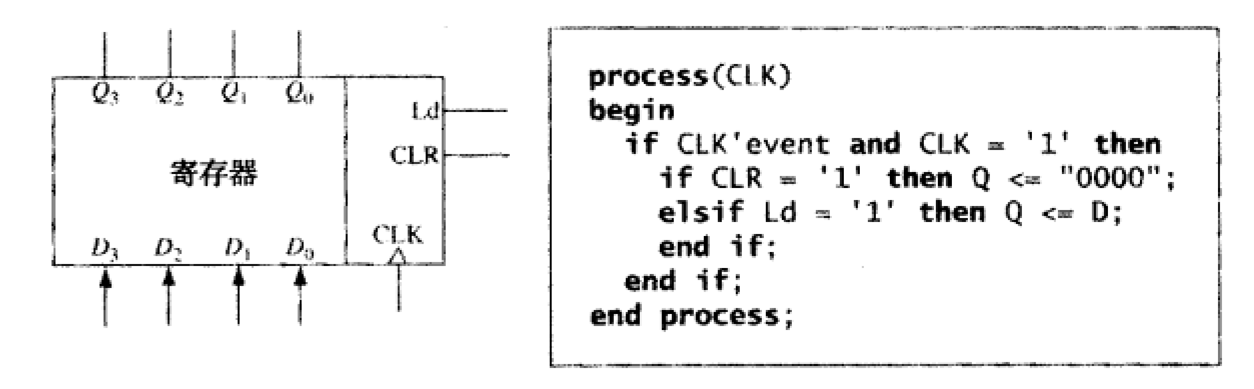


图 2.42　具有同步清零和载入端的寄存器

接着，我们将用 VHDL 的进程描述左移寄存器。图 2.43 中寄存器与图 2.42 中寄存器很相似，只是我们在图 2.43 中的寄存器上添加了一个左移控制输入（*LS*）。当 *LS* = '1'时，寄存器的内容向左移一位，且最右边的一位被置为 R_{in}。移位是通过提取 *Q* 的最右边的三位 `Q(2 downto 0)`，再把它和 R_{in} 拼接而完成。例如，如果 *Q* = "1101"且 R_{in} = '0'，则 `Q(2 downto 0)&Rin` = "1010"，且此值在时钟 *CLK* 上升沿到来时被置回到寄存器 *Q* 中。该代码还暗示，如果 *CLR* = *Ld* = *LS* = '0'，则 *Q* 保持不变。

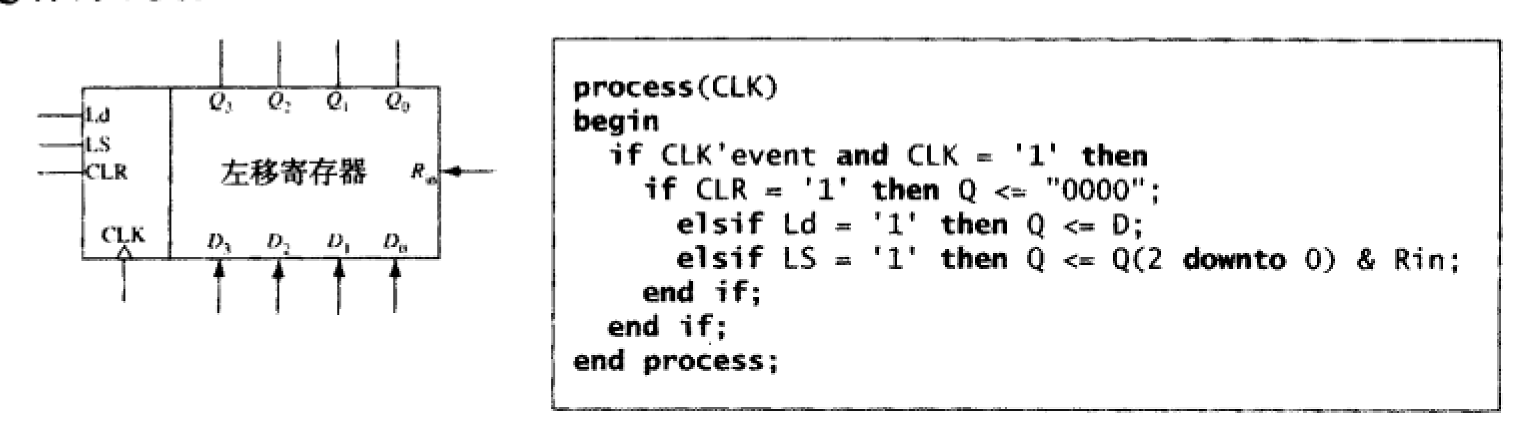


图 2.43　具有同步清零和载入端的左移寄存器

图 2.44 为一个简单的同步计数器。在时钟的上升沿到来时，如果 *ClrN* = '0'，则计数器将被清零；当 *ClrN* = *En* = '1'时，它才计数。在本例中，信号 *Q* 是存储在计数器中的 4 位数值。由于针对 bit-vector 类型没有定义加法操作，所以我们把 *Q* 说明为无符号类型。然后使用 IEEE.numeric_bit 包集合中的重载运算符"+"对计数器进行加 1 操作。语句 `Q<=Q+ 1;` 实现对计数器加 1 的操作。当计数器在状态"1111"时，下一次计数使计数器会回到状态"0000"。

```
signal Q: unsigned (3 downto 0);
------------
process (CLK)
begin
  if CLK'event and CLK = '1' then
    if ClrN = '0' then Q <= "0000";
    elsif En = '1' then Q <= Q + 1;
    end if;
  end if;
end process;
```

图 2.44　一个简单的同步计数器的 VHDL 程序

现在，我们要编写一个通用计数器 74163 的 VHDL 程序模块。74163 是一个 4 位完全同步清

二进制计数器，由 TTL 逻辑和 COMS 逻辑两种产品。虽然现在我们已经很少用于新设计中，但它作为一种基本的计数器在许多 CAD 设计库中都能找到。除了有计数功能外，它还具有并行清零和置数功能，而且所有这些操作都与时钟同步，并且在时钟上升沿发生状态改变。该计数器的框图如图 2.45 所示。

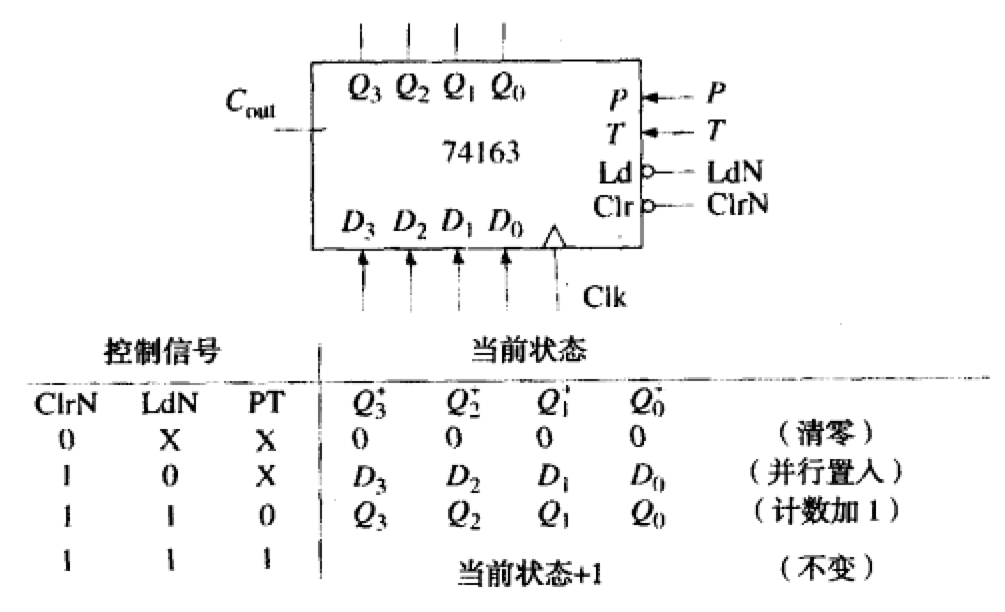
这种计数器有 4 个控制输入：*ClrN, LdN, P* 和 *T*。输入 *P* 和 *T* 是计数功能使能信号。计数器的操作如下：

1. 如果 *ClrN* = '0'，则在时钟上升沿到来时，所有触发器清零。
2. 如果 *ClrN* = '1'且 *LdN* = '0'，则在时钟上升沿到来时，*D* 输入并行载入到触发器中。
3. 如果 *ClrN* = *LdN* = '1'且 *P* = *T* = '1'，则在时钟上升沿到来时，计数器开始计数且每次加 1。

如果 *T* = '1'，则计数器在状态 15 产生生成一个进位（C_{out}），所以

$$C_{out} = Q_3Q_2Q_1Q_0T$$

图 2.45 所示真值表归纳了计数器的操作。注意，*ClrN* 优先于置数和计数使能，也就是说，当 *ClrN* = '0'时，不管 *LdN, P* 和 *T* 为何值，都会对计数器清零。类似的，*LdN* 优先于计数使能。74163 的 *ClrN* 输入称为一个同步清零输入，这是因为计数器清零时是与时钟同步的，当时钟脉冲没有到来之前清零不会发生。



控制信号			当前状态				
ClrN	LdN	PT	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	（清零）
1	0	X	D_3	D_2	D_1	D_0	（并行置入）
1	1	0	Q_3	Q_2	Q_1	Q_0	（计数加 1）
1	1	1	当前状态+1				（不变）

图 2.45 74163 计数器的操作

计数器的 VHDL 描述见图 2.46。*Q* 表示组成计数器的 4 个触发器。只要 *Q* 发生变化，则计数器输出 Q_{out} 就发生改变。当 *Q* 或 *T* 改变时，都会计算进位输出；进程中的第一个 if 语句是用来检测时钟上升沿的。由于清零操作优先于置数和计数操作，所以下一个 if 语句先检测 *ClrN*。由于置数优先于计数操作，所以下一个检测的是 *LdN*。最后，如果 *P* 和 *T* 均为 1，计数器就开始计数。由于 *Q* 是无符号类型的，所以我们可以使用 IEEE.numeric_bit 包集合中的重载运算符"+"对计数器加 1。如果 *Q* 是 bit-vector 类型，则表达式 *Q*+1 是不合法的，因为 bit-vector 类型没有定义加法运算。

为了测试此计数器，我们把两个 74163 级联组成一个 8 位加法计数器（参见图 2.47）。当右边的计数器状态为 1111，且 T_1='1'时，*Carry1* = '1'。这时左边的计数器，如果 *P*='1'，则 *P·T* = '1'。如果 *PT*='1'，则当下一个时钟上升沿到来时，右边的计数器加 1 变为 0000，同时左边的计数器也加 1。图 2.48 给出了两个 74163 级联组成 8 位加法计数器的 VHDL 代码。在该代码中，c74163

模块作为元件被调用了两次。为了方便地读出计数器的输出，我们定义一个整数类型（integer）信号为 *Count*，它与计数器 8 位二进制数是等价的。代码中 to_integer 函数把无符号矢量转换为一个整数。

```
-- 74163 FULLY SYNCHRONOUS COUNTER

library IEEE;
use IEEE.numeric_bit.all;

entity c74163 is
  port(LdN, ClrN, P, T, Clk: in bit;
       D: in unsigned(3 downto 0);
       Cout: out bit; Qout: out unsigned(3 downto 0));
end c74163;

architecture b74163 of c74163 is
signal Q: unsigned(3 downto 0);    -- Q is the counter register
begin
  Qout <= Q;
  Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
  process(Clk)
  begin
    if Clk'event and Clk = '1' then    -- change state on rising edge
      if ClrN = '0' then Q <= "0000";
      elsif LdN = '0' then Q <= D;
      elsif (P and T) = '1' then Q <= Q + 1;
      end if;
    end if;
  end process;
end b74163;
```

图 2.46　74163 计数器程序

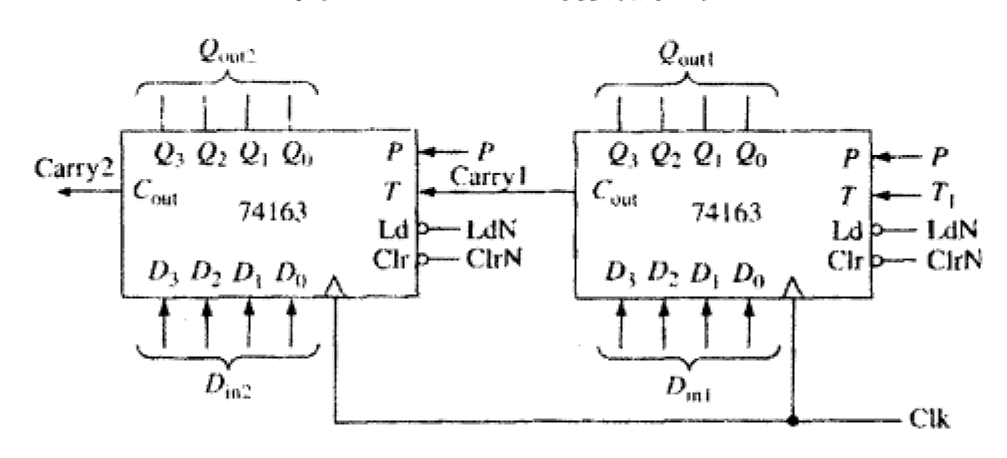


图 2.47　两个 74163 级联组成的 8 位加法计数器

```
--Test module for 74163 counter

library IEEE;
use IEEE.numeric_bit.ALL;

entity eight_bit_counter is
  port(ClrN, LdN, P, T1, Clk: in bit;
       Din1, Din2: in unsigned(3 downto 0);
       Count: out integer range 0 to 255;
       Carry2: out bit);
end eight_bit_counter;

architecture cascaded_counter of eight_bit_counter is
component c74163
  port(LdN, ClrN, P, T, Clk: in bit;
       D: in unsigned(3 downto 0);
       Cout: out bit; Qout: out unsigned(3 downto 0));
end component;

signal Carry1: bit;
signal Qout1, Qout2: unsigned(3 downto 0);
begin
  ct1: c74163 port map (LdN, ClrN, P, T1, Clk, Din1, Carry1, Qout1);
  ct2: c74163 port map (LdN, ClrN, P, Carry1, Clk, Din2, Carry2, Qout2);
  Count <= to_integer(Qout2 & Qout1);
end cascaded_counter;
```

图 2.48　两个 74163 级联构成的 8 位计数器的 VHDL 程序

现在把左移寄存器（参见图 2.43）的 VHDL 代码进行综合。在开始综合之前，我们必须先

给出目标器件（例如特定 FPGA 或 CPLD 等），这样综合器才知道哪些元件是可用的。假设目标器件为 CPLD 或 FPGA，它包含带有时钟使能端的 D 触发器（D-CE 触发器）。Q 和 D 均为 4 位矢量。由于 Q 的更新紧随着"CLK'event and CLK = '1' then"之后，所以 Q 必须是由 4 个触发器构成的寄存器，分别记为 Q_3, Q_2, Q_1, Q_0。由于当 *Clr*、*Ld* 或 *Ls* 为'1'时，触发器才发生状态改变，所以我们把时钟使能连接到一个或门（OR）的输出，其输出为 $Clr + Ld + Ls$。然后我们把一些逻辑门与输入 D 连接，以便选择将要被置入触发器的数据。如果 Clr = '0'且 Ld = '1'，则在时钟上升沿到来时，D 被置入到触发器中；如果 $Clr = Ld$ = '0'且 Ls = '1'，则 Q_2 被置入到 Q_3 中，Q_1 被置入到 Q_2 中，依次类推。图 2.49 为前两个触发器的逻辑电路。如果 Clr = '1'，则 D 触发器输入均为 0 且寄存器清零。

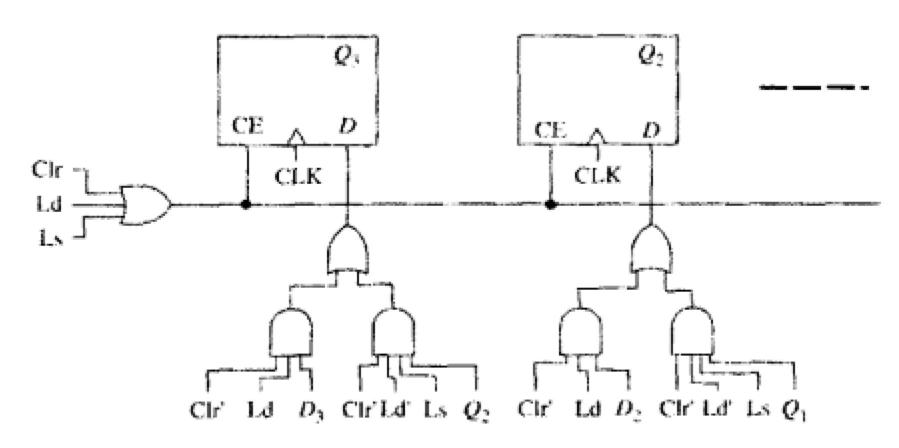


图 2.49　图 2.43 左移寄存器 VHDL 程序的综合

VHDL 综合器不能对延迟进行综合。格式为“`after` 时间表达式”的语句在大多数综合器中均被忽略，但是有些综合器需要去掉 **after** 句法。虽然信号的初始值可以在端口和信号说明中指定，但是这些初始值均被综合器忽略。如果该硬件电路必须要设置特定的初始值，则必须提供一个复位信号。否则，硬件电路的初始状态是不定的，其功能不一定正确。当有整数信号进行综合时，该整数在硬件上用等效的二进制数来表示的。如果整数的取值范围没有给定，则综合器将假设为最大位数，一般为 32 位。比如，

```
signal count: integer range 0 to 7
```

该语句将生成一个 3 位计数器，但是

```
signal count: integer
```

会生成一个 32 位计数器。

VHDL 信号将保持其当前值，直到其发生改变。这样，在综合时会生成一个不希望的锁存器。比如，在组合进程中，语句

```
if X = '1' then B<= 1; end if;
```

将会生成锁存器。这样，当 X 变为'0'时，仍旧保持 B 的值。为了防止在组合进程中生成不必要的锁存器，必须在 **if** 语句中总使用 **else** 语句。例如，语句

```
if X = '1' then B<= 1; else B<= 0; end if;
```

将会生成一个 MUX，控制 B 的取值为 1 或 0。

2.15　VHDL 的行为和结构描述方式

任何电路或设备都可以在不同抽象层面上表示。图 2.50 给出了 NAND（与非）门的各种不同层面上表示。一听到 NAND 这个词的时候，由于设计层面不同，不同设计者的脑海中就会出现 NAND 门设备的不同表示形式。有些人可能想到行为描述的框图，如图 2.50(a)所示；有些人会想到 CMOS 7400 芯片中的 4 个门，如图 2.50(b)所示；在逻辑层面设计者会想到 NAND 门的逻辑符号，如图 2.50(c)所示；晶体管设计者会想到生成 NAND 的晶体管电路，如图 2.50(d)所示；物理层设计者会想到 NAND 门的布局，如图 2.50(e)所示。所有这些图，虽然都表示同一个 NAND 门，但是它们在具体表达细节程度上不一样。

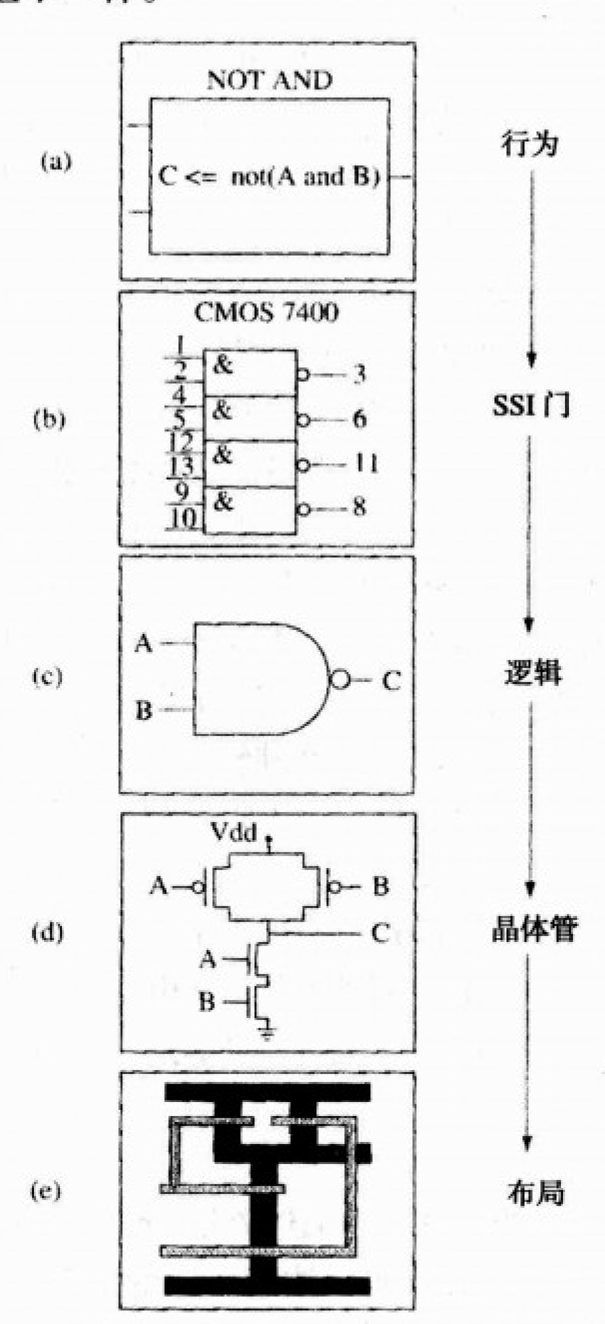


图 2.50　NAND 门的各种不同表达方式

就像 NAND 门可以用不同方式表示一样，任何逻辑电路在不同层面都有不同的表示形式。图 2.51 是逻辑表达式 $F = ab + bc$ 的行为描述方式，图 2.52 是该表达式的两种等价的结构描述方式。在图 2.51 中抽象描述的函数功能，可以通过不同的方法实现，比如可以使用两个 AND 门和一个 OR 门实现，也可以使用 3 个 NAND 门实现。虽然图 2.52(a)和图 2.52(b)分别给出了两种不同的结构表示，但是它们具有相同的行为功能。结构描述方式更具体，而行为描述方式则是更抽象的上一层面上的描述。

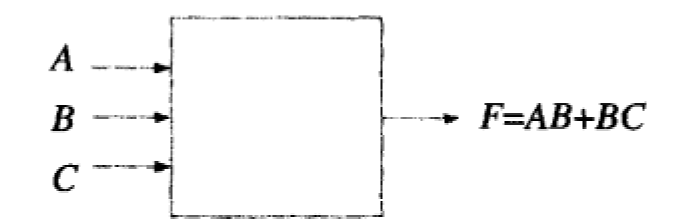
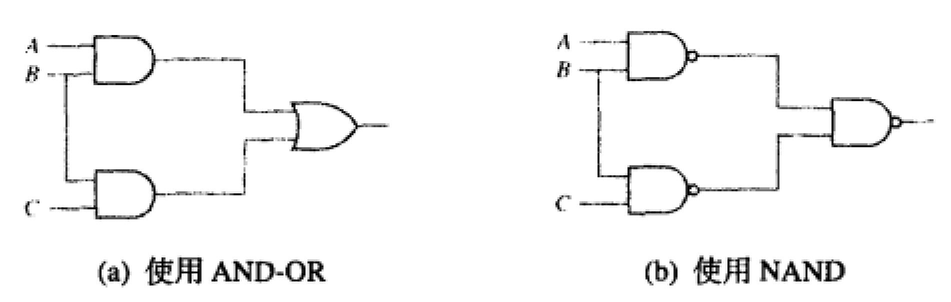


图 2.51　输入为 A, B, C，输出为 $F = AB + BC$ 的框图



(a) 使用 AND-OR　　(b) 使用 NAND

图 2.52　$F = \mathrm{AB} + \mathrm{BC}$ 的两种实现

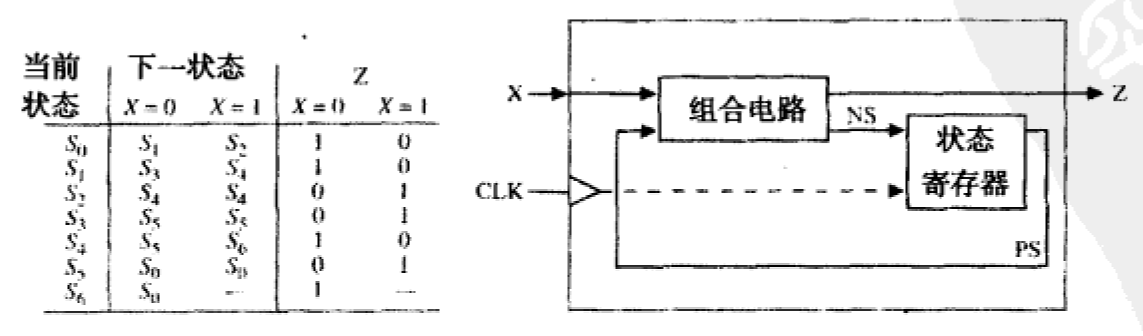
你注意到同一个电路可以在不同的层面上描述。同样，VHDL 语言也允许我们在不同的抽象层面上进行设计描述。最常用的有行为模型、数据流模型（寄存器转移语言[Register transfer language，RTL]）和结构模型。VHDL 的行为模型是在较高抽象层面上对电路或系统进行描述，而不用涉及任何特定的结构或实现技术，只对整体行为进行描述。相反，在结构模型中，必须清楚地描述所用的元件和元件之间的互连关系。结构模型的描述可以很具体，都可以给出特定库或包集合的特定的逻辑门或触发器。所以 VHDL 的结构模型是抽象层面上比较低的。除了纯粹的行为层面和结构层面之外，VHDL 代码也可以在中间抽象层面上进行描述，这就是数据流层面或者 RTL 层面。几十年来，寄存器转移语句（Register transfer languages）主要用于描述同步系统的行为。这些系统均可以视为由寄存器和对其进行置入和操作的控制逻辑电路组成的。在数据流模型中，对数据通路和控制信号都进行详细的描述。系统的工作主要体现在这些寄存器之间的数据转移。

如果在较高的层面进行设计，则为了硬件实现，必须把设计转换到较低的层面上来。早期的设计自动化中，没有足够自动的软件可以完成这种转换，因此必须在低层面上对设计进行详细的描述。当时可以采用图形输入技术或者较低层面上抽象描述方法。但现在，我们可以使用综合工具有效地把行为描述设计转换到目标技术。

行为描述设计和结构描述设计经常是在一起使用的。一个设计不同的部分通常使用不同的描述方式。现代的自动设计工具都可以生成有效的逻辑和算术电路，所以这些设计中很大一部分是在行为层面上的设计的。但是，存储器结构通常需要进行人工优化，依赖于特殊设计，不是靠自动综合。

2.15.1　时序机建模

本节中我们将讨论几种时序机的 VHDL 描述方法。首先，我们根据如图 2.53 给出的状态表写出 Mealy 时序电路的行为描述 VHDL 代码（注意，这就是第 1 章中介绍过的 BCD 码到余 3 码的码转换器）。该状态机的框图也在图 2.53 中给出。这一框图可以用于写出 VHDL 实体说明部分代码。请注意，当前状态和下一状态对外部是不可见的。



当前状态	下一状态		Z	
	$X=0$	$X=1$	$X=0$	$X=1$
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

图 2.53　时序机的状态表和框图

该时序机可以用多种方法进行模拟。一种方法是用两个进程分别表示电路的两个部分。一个进程用来模拟电路的组合逻辑部分，用以产生下一状态信息和输出；另一个进程则用来模拟状态寄存器，在适当的时钟沿更新状态。图 2.54 为给出了模拟该 Mealy 时序机的 VHDL 代码。第一个进程表示组合电路。在行为描述层，我们将用整数信号表示当前状态和下一状态，其初始值为 0。请注意，该初始值只对仿真有意义。因为电路的输出 Z 和 *Nextstate* 的变化，都依赖于 *State* 和 *X* 中的任何一个变化，所以在敏感信号表中包括 *State* 和 *X*。CASE 语句用于检测 *State* 值，并根据 *X* 值，更新 Z 和 *Nextstate* 的值。第二个进程则表示状态寄存器。当时钟上升沿到来时，*State* 更新为当前 *Nextstate* 的值，所以时钟信号 *CLK* 出现在敏感信号表中。如果按如下方式编程，则第二个进程可以正确地仿真：

```
process(CLK)                -- 状态寄存器
begin
   If CLK = '1' then        -- 时钟上升沿
     State <=Nextstate;
   end if;
end process;
```

但是为了用边沿触发的触发器进行综合，必须使用如下的属性语句 `clk'event`：

```
process(CLK)                              -- 状态寄存器
begin                                     -- 综合
   if CLK'event and CLK = '1' then        -- 时钟上升沿
     State <=Nextstate;
   end if;
end process;
```

图 2.54 中的 *State* 是一个取值范围为 0 ~ 6 的整数。语句 **when others = >null** 实际上多余的，因为在 **case** 语句中明确地列出了输出和下一状态 *State* 的所有可能取值。但是当任何 **if** 语句中的 **else** 被省略或对于 *State* 的所有可能值所做的操作都没有明确给出时，我们就需要使用该语句。代码中的 null 表示不做任何操作。在这里用它是合理的，因为 *State* 的其他取值（others）不会出现。如果 **else** 语句被省略或有一个条件没有给出相应的操作，则在综合时一般生成锁存器。

```
-- This is a behavioral model of a Mealy state machine (Figure 2-53)
-- based on its state table. The output (Z) and next state are
-- computed before the active edge of the clock. The state change
-- occurs on the rising edge of the clock.

entity Code_Converter is
  port(X, CLK: in bit;
       Z: out bit);
end Code_Converter;

architecture Behavioral of Code_Converter is
signal State, Nextstate: integer range 0 to 6;
begin
  process(State, X)              -- Combinational Circuit
  begin
    case State is
      when 0 =>
        if X = '0' then Z <= '1'; Nextstate <= 1;
```

图 2.54　余 3 码转换器的行为描述方式 VHDL 程序

```
        else Z <= '0'; Nextstate <= 2; end if;
      when 1 =>
        if X = '0' then Z <= '1'; Nextstate <= 3;
        else Z <= '0'; Nextstate <= 4; end if;
      when 2 =>
        if X = '0' then Z <= '0'; Nextstate <= 4;
        else Z <= '1'; Nextstate <= 4; end if;
      when 3 =>
        if X = '0' then Z <= '0'; Nextstate <= 5;
        else Z <= '1'; Nextstate <= 5; end if;
      when 4 =>
        if X = '0' then Z <= '1'; Nextstate <= 5;
        else Z <= '0'; Nextstate <= 6; end if;
      when 5 =>
        if X = '0' then Z <= '0'; Nextstate <= 0;
        else Z <= '1'; Nextstate <= 0; end if;
      when 6 =>
        if X = '0' then Z <= '1'; Nextstate <= 0;
        else Z <= '0'; Nextstate <= 0; end if;
      when others => null;           -- should not occur
    end case;
  end process;

  process(CLK)                       -- State Register
  begin
    if CLK'EVENT and CLK = '1' then -- rising edge of clock
      State <= Nextstate;
    end if;
  end process;
end Behavioral;
```

图 2.54（续） 余 3 码转换器的行为描述方式 VHDL 程序

可以用来测试图 2.54 的仿真器的命令如下：

```
add wave CLK X State NextState Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

第一条命令指定要输出波形的信号。下一条命令定义一个周期为 200 ns 的时钟信号，该时钟信号初始值为 0，在 100 ns 后变为 1。命令语句的格式为

```
force  信号名  v1   t1, v2  t2, ...
```

它表示信号在 `t1` 时刻赋值为 `v1`，`t2` 时刻赋值为 `v2` 等。*X* 在 0 时刻为'0'，在 350 ns 时变为'1'，又在 550 ns 时变为'0'等，依次类推。输入 *X* 对应于序列 0010 1001，只有 *X* 发生变化的时刻值被指定。上面命令语句的执行结果如图 2.55 所示。

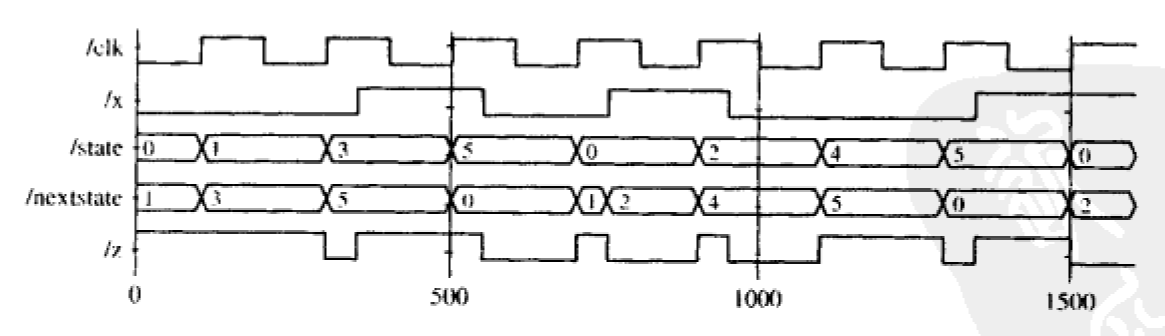


图 2.55 余 3 码转换器的模拟器输出波形

在第 1 章中，我们手工设计了该状态机（参见图 1.26）。该电路由 7 个 3 输入 NAND 门，3 个 D 触发器和 1 个非门构成。图 2.54 给出的行为描述方式 VHDL 程序代码不一定能综合生成与图 1.26 完全相同的电路。实际上，当我们用 Xilinx ISE 工具对该程序进行综合时，我们得到的电路是由 7 个 D 触发器、15 个 2 输入 AND 门、3 个 2 输入 OR 门和 1 个 7 输入 OR 门构成的。显然，Xilinx 综合工具使用默认的 one-hot 状态赋值，而不是采用编码赋值。One-hot 设计是 FPGA

的常用方法，它需要使用大量的触发器。

```
-- This is a behavioral model of the Mealy state machine for BCD to
-- Excess-3 Code Converter based on its state table. The state change
-- occurs on the rising edge of the clock. The output is computed by a
-- conditional assignment statement whenever State or Z changes.

entity Code_Converter2 is
  port(X, CLK: in bit;
       Z: out bit);
end Code_Converter2;

architecture one_process of Code_Converter2 is
signal State: integer range 0 to 6 : = 0;
begin
  process(CLK)
  begin
    if CLK'event and CLK = '1' then
      case State is
        when 0 = >
          if X = '0' then State <= 1; else State <= 2; end if;
        when 1 = >
          if X = '0' then State <= 3; else State <= 4; end if;
        when 2 = >
          State <= 4;
        when 3 = >
          State <= 5;
        when 4 = >
          if X = '0' then State <= 5; else State <= 6; end if;
        when 5 = >
          State <= 0;
        when 6 = >
          State <= 0;
      end case;
    end if;
  end process;
  Z <= '1' when (State = 0 and X = '0') or (State = 1 and X = '0')
              or (State = 2 and X = '1') or (State = 3 and X = '1')
              or (State = 4 and X = '0') or (State = 5 and X = '1')
              or State = 6
         else '0';
end one_process;
```

图 2.56　使用单一进程实现码转换器的行为描述方式 VHDL 程序

图 2.56 给出了码转换器的行为描述的另一种 VHDL 模型，该模型只使用了一个进程而不是两个进程。该电路的下一状态没有显式计算，但是在时钟上升沿，根据下一状态的可能值直接对状态寄存器进行更新。由于只要状态或 X 发生变化，则 Z 就发生变化，所以 Z 不在时钟进程中进

行计算。我们用条件赋值语句对 Z 进行计算。如果 Z 在时钟进程中更新，则需要一个触发器将保存 Z 值，并且 Z 的更新时间不对。一般来说，描述状态机时，使用两个进程要比使用一个进程要好。这是因为前者与实现硬件更接近，该硬件电路是由一个组合逻辑电路和一个状态寄存器组成的。

我们也可以使用数据流描述方式对该 Mealy 机进行模拟。图 2.57 给出的 VHDL 的数据流模型是基于下一状态和输出的逻辑表达式，我们曾在第 1 章推导过该表达式（参见图 1.25）。触发器的更新是在进程中进行的，*CLK* 在该进程的敏感信号表中。每当时钟上升沿出现时，Q_1, Q_2 和 Q_3 都赋予新值。从时钟上升沿到触发器输出有变化的传输延迟为 10 ns。在进程中，即使赋值语句还是按照顺序进行的，但是 Q_1, Q_2 和 Q_3 值的更新调度时间是相同的，即 $T+\Delta$ 时刻，其中 T 表示时钟上升沿出现时刻。这样，前一时刻 Q_1 的值将用于计算下一时刻 Q_2 的值（Q_2^+）；前一时刻的 Q_1, Q_2 和 Q_3 的值将用于计算下一时刻 Q_3 的值（Q_3^+）。Z 的并发语句，保证其更新紧随着 X 或 Q_3 值的任何变化。两个门电路的总时延为 20 ns。注意，为了在这一层面上进行 VHDL 建模，我们需要进行状态赋值，推导出下一状态表达式等。相反，在行为描述层面上，我们只需知道状态表，就足以建立 VHDL 模型。

```
-- The following is a description of the sequential machine of
-- the BCD to Excess-3 code converter in terms of its next state
-- equations. The following state assignment was used:
-- S0-->0; S1-->4; S2-->5; S3-->7; S4-->6; S5-->3; S6-->2

entity Code_Converter is
  port(X, CLK: in bit;
       Z: out bit);
end Code_Converter;

architecture Equations of Code_Converter is
signal Q1, Q2, Q3: bit;
begin
  process(CLK)
   begin
     if CLK = '1' and CLK'event then     -- rising edge of clock
       Q1 <= not Q2 after 10 ns;
       Q2 <= Q1 after 10 ns;
       Q3 <= (Q1 and Q2 and Q3) or (not X and Q1 and not Q3) or
             (X and not Q1 and not Q2) after 10 ns;
     end if;
   end process;
   Z <= (not X and not Q3) or (X and Q3) after 20 ns;
 end Equations;
```

图 2.57 基于方程的时序机模型

前面 Mealy 时序机的另一个 VHDL 模型是结构模型，对电路中的门和触发器等元件进行描述。图 2.58 给出了图 1.26 电路的 VHDL 结构模型。注意到，为了得到图 2.58 的模块，设计者必须手工设计以得到门电路层面上电路图。在第 1 章中，该时序机需要 7 个与非门、3 个 D 触发器和 1 个非门。当需要最基本的组成元件（如门和触发器）时，这些元件都可以在另一单独的 VHDL 模块中定义。根据所用的 CAD 工具的不同，我们可以把那些常用元件模块写进同一个文件中（像 VHDL 的主程序），或者作为单独的文件插入到一个 VHDL 项目（project）中。图 2.58 程序代码需要的模块有：DFF, Nand3, Nand2 和反相器。CAD 工具的包集合中可能含有类似的元件。如果使用这样的包集合，则我们必须使用正确的元件名和端口映射语句与包集合中元件的输入输出信号相匹配。DFF 模块如下所示：

```
--D Flip-Flop
entity DFF is
  port (D, CLK: in bit;
        Q: out bit;  QN: out bit : = '1');
```

```
-- initialize QN to '1' since bit signals are initialized to '0'
end DFF;
architecture SIMPLE of DFF is
begin
  process (CLK)             -- process is executed when CLK changes
  begin
    if CLK'event and CLK = '1' then             -- rising edge of clock
      Q  <= D after 10 ns;
      QN  <= not D after 10 ns;
    end if;
  end process;
end SIMPLE;
```

Nand3 模块如下所示：

```
--3 input NAND gate
entity Nand3 is
  port (A1, A2, A3: in bit; Z: out bit);
end Nand3;
architecture concur of Nand3 is
begin
Z <= not (A1 and A2 and A3)after 10 ns;
end concur;
```

```
-- The following is a STRUCTURAL VHDL description of
-- the circuit to realize the BCD to Excess-3 code Converter.
-- This circuit was illustrated in Figure 1-20.
-- Uses components NAND3, NAND2, INVERTER and DFF
-- The component modules can be included in the same file
-- or they can be inserted as separate files.

entity Code_Converter is
  port(X,CLK: in bit;
       Z: out bit);
end Code_Converter;

architecture Structure of Code_Converter is
component DFF
  port(D, CLK: in bit; Q: out bit; QN: out bit := '1');
end component;
component Nand2
  port(A1, A2: in bit; Z: out bit);
end component;
component Nand3
  port(A1, A2, A3: in bit; Z: out bit);
end component;
component Inverter
  port(A: in bit; Z: out bit);
end component;
signal A1, A2, A3, A5, A6, D3: bit;
signal Q1, Q2, Q3: bit;
signal Q1N, Q2N, Q3N, XN: bit;
begin
  I1:  Inverter port map (X, XN);
  G1:  Nand3 port map (Q1, Q2, Q3, A1);
  G2:  Nand3 port map (Q1, Q3N, XN, A2);
  G3:  Nand3 port map (X, Q1N, Q2N, A3);
  G4:  Nand3 port map (A1, A2, A3, D3);
  FF1: DFF port map (Q2N, CLK, Q1, Q1N);
  FF2: DFF port map (Q1, CLK, Q2, Q2N);
  FF3: DFF port map (D3, CLK, Q3, Q3N);
  G5:  Nand2 port map (X, Q3, A5);
  G6:  Nand2 port map (XN, Q3N, A6);
  G7:  Nand2 port map (A5, A6, Z);
end Structure;
```

图 2.58　时序机的结构模型

除了输入端口数不同外，Nand2 和反相器模块与 Nand3 模块大体相同。我们假设了每个元件存在 10 ns 的延迟，该延迟时间可以很容易进行修改以反应所用硬件的实际延迟。

由于 Q_1, Q_2, Q_3 的初始值为'0'，所以与它们互补的触发器输出 Q_1N, Q_2N, Q_3N 的初值为'1'；G_1 为三输入与非门，它的三个输入为 Q_1, Q_2, Q_3，输出为 A_1；FF_1 是 D 触发器，其输入 D 与 Q_2N 相连。执行下面的仿真器命令文件，就得到如图 2.59 所示波形，该波形图与图 1.39 的波形很相似。

```
add wave CLK X Q1 Q2 Q3 Z
force CLK 0 0, 1 100    -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

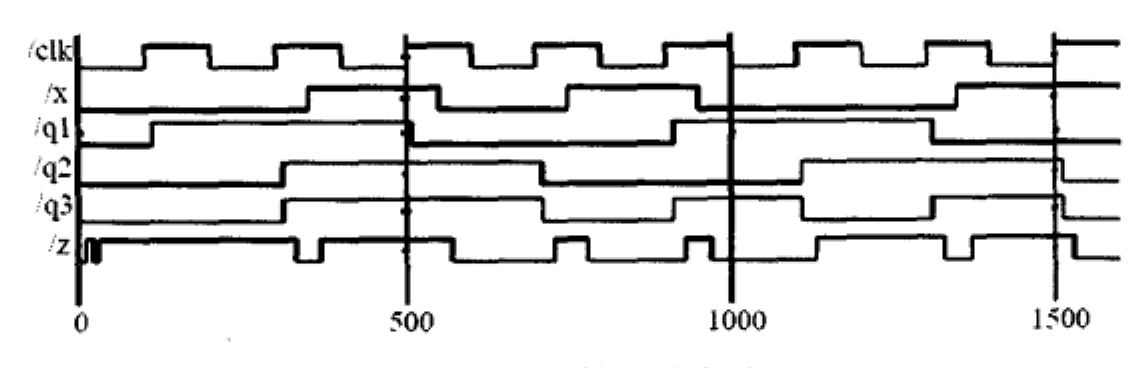


图 2.59　码转换器波形

如果我们综合这一结构描述的 VHDL 代码，则结果我们想要的电路完全相同，这一电路是由 3 个 D 触发器、3 个 2 输入 NAND 门和 4 个 3 输入 NAND 门组成的。综合图 2.54 后得到的电路，则由 7 个触发器、15 个 2 输入 AND 门、3 个 2 输入 OR 门和 1 个 7 输入 OR 门组成。请比较这两个电路。当设计者对所有的元件和它们之间的连线都给出时，综合工具也不必再理会或者去"猜"。

对 C 语言代码进行汇编内联优化时，也会出现类似的情况。在优化生成汇编代码时，我们必须准确描述所要的微处理器指令顺序，编译器就给出你所要的结果。同样，综合器不必把设计者所写的所有的结构描述都进行转换处理，它只需确定设计者给出的电路结构中的硬件。有些优化工具可以对你设计的电路进行优化。一般当你使用结构描述方式时，你对最后生成电路的控制度是比较大的。但是由于结构描述模型需要进行状态赋值，需要推导下一状态表达式等，所以编写结构描述方式的 VHDL 代码需要做出更多的努力。要想在 IC 市场上取得成功，则必须要注意时效性（time-to-market）。因此，设计者通常使用行为描述模型以便节省时间，使芯片早日投放市场。另外，CAD 工具在过去的 20 年中变得越发成熟，大多数综合工具已经能给出有效的算术和逻辑电路。

2.16　变量、信号和常数

在前面的 VHDL 代码中，我们只用过信号（signals），还没用变量（variables）。VHDL 语言中也提供与其他通用高级程序语言类似的变量。变量可以用于进程的本地存储单元，也可以用于过程和函数中（我们还没有介绍）。本节介绍的大部分内容只与仿真有关。

变量定义的说明语句的格式为

variable　变量名：数据类型　[:=初始值]；

变量是一个局部量，它必须在所在的进程中定义说明（共享变量除外，本书中不对其进行介绍）。相反，信号必须在进程之外定义说明。在结构体的开始定义的信号，在该结构体的任何地

方都可以用。信号定义说明语句的格式为：

```
signal  信号名: 数据类型 [:=初始值];
```

变量通过赋值语句进行更新，变量赋值语句的语法格式为

```
变量名:=表达式;
```

当执行该语句时，变量的赋值几乎没有延时，甚至连Δ延时也没有。相反，我们考虑下面的信号赋值语句格式：

```
信号名 <= 表达式 [after 延迟];
```

当执行语句时，给信号安排一个调度，经过延迟后信号才发生变化。若没有给定延时时间，则经过Δ延时后才更新。

下面的表达是错误的：

```
变量名 <= 表达式 [after 延迟];
```

如何使用信号和变量：如果要模拟的物理量与电路中某些物理信号相对应，则应使用信号。如果要模拟的物理量只是一个暂时值，是为了编程方便，则使用变量就足够了。对变量表示的值不会出现在电路中的任何物理线路上。如果想要这些值出现，则必须使用信号。

从图 2.60 和图 2.61 的例子说明了在一个进程中使用变量或者信号的区别。变量必须在进程语句内部说明和初始化，而信号必须在进程外部定义说明和初始化。图 2.60 中，如果在 time = 10 时 *trigger* 变化，Var1, Var2, Var3 将按前后顺序计算并且立刻更新，随后用更新后的值计算出 Sum 的结果。这一计算顺序为：Var1 = 2 + 3 = 5, Var2 = 5, Var3 = 5, 然后计算出 Sum = 5 + 5 + 5 = 15。由于 Sum 是信号，所以它有Δ延时，即在 10+Δ时刻 Sum = 15。总之，变量的使用与其他编程语言中的使用规则相同，但信号的更新却一定需要时间延迟。图 2.61 中，如果在 time = 10 时 *trigger* 变化，Sig1, Sig2, Sig3 和 Sum 都在 time = 10 时就进行计算，但是信号的值在 10+Δ时刻才更新，计算 Sig2 和 Sig3 的所用的 Sig1 和 Sig2 的值是以前的值。所以，在 time=10+Δ时刻，Sig1 = 5, Sig2 = 1, Sig3 = 2, Sum = 6。

```
entity dummy is
end dummy;

architecture var of dummy is
signal trigger, sum: integer:=0;
begin
    process
    variable var1: integer:=1;
    variable var2: integer:=2;
    variable var3: integer:=3;
    begin
        wait on trigger;
        var1 := var2 + var3;
        var2 := var1;
        var3 := var2;
        sum <= var1 + var2 + var3;
    end process;
end var;
```

ns	delata	trigger	Vra1	Var2	Var3	Sum
0	+0	0	1	2	3	0
0	+1	0	1	2	3	0
10	+0	1	5	5	5	0
10	+1	1	5	5	5	15

图 2.60　含有变量的进程及相应的仿真输出

```
entity dummy is
end dummy;

architecture sig of dummy is
     signal trigger, sum: integer:=0;
     signal sig1: integer:=1;
     signal sig2: integer:=2;
     signal sig3: integer:=3;
begin
     process
     begin
       wait on trigger;
       sig1 <= sig2 + sig3;
       sig2 <= sig1;
       sig3 <= sig2;
       sum <= sig1 + sig2 + sig3;
     end process;
end sig;
```

ns	delata	trigger	Vra1	Var2	Var3	Sum
0	+0	0	1	2	3	0
0	+1	0	1	2	3	0
10	+0	1	1	2	3	0
10	+1	1	5	1	2	6

图 2.61　含有信号的进程以及相应的仿真输出

在仿真时，初始化使进程执行一次，当遇到 wait 语句时就停止。因此仿真的输出直接依赖于 wait 语句所在位置：进程的开始还是在进程的结尾。是否使用了敏感信号表，也影响仿真输出。图 2.62 和图 2.63 说明了多种可能性。请记住，这些差别对 VHDL 综合并不重要，它们只对 VHDL 行为仿真产生影响。

```
entity dummy is
end dummy;

architecture var of dummy is
signal trigger, sum: integer: = 0;
begin
  process(trigger)
  variable var1: integer: = 1;
  variable var2: integer: = 2;
  variable var3: integer: = 3;
  begin
    var1 : = var2 + var3;
    var2 : = var1;
    var3 : = var2;
    sum <= var1 + var2 + var3;
  end process;
end var;
```

ns	delata	trigger	Vra1	Var2	Var3	Sum
0	+0	0	1	2	3	0
0	+1	0	5	5	5	15
10	+0	1	10	10	10	15
10	+1	1	10	10	10	30

图 2.62　含有变量的进程以及相应的仿真输出

```
entity dummy is
end dummy;

architecture sig of dummy4 is
signal trigger, sum: integer: = 0;
signal sig1: integer: = 1;
signal sig2: integer: = 2;
signal sig3: integer: = 3;
begin
  process(trigger)
```

ns	delata	trigger	Sig1	Sig2	Sig3	Sum
0	+0	0	1	2	3	0
0	+1	0	5	1	2	6
10	+0	1	5	1	2	6
10	+1	1	3	5	1	8

图 2.63　含有信号的进程以及相应的仿真输出

```
  begin
    sig1 <= sig2 + sig3;
    sig2 <= sig1;
    sig3 <= sig2;
    sum <= sig1 + sig2 + sig3;
  end process;
end sig;
```

图 2.63（续）　含有信号的进程以及相应的仿真输出

2.16.1　常数

像变量一样，常数的使用也为编程带来了方便。

常数定义说明语句的格式为

```
constant  常数名：数据类型:= 恒定值;
```

例如，数值为 5 ns，数据类型为 time 的常数 delay1 可以定义为

```
constant   delay1: time: = 5 ns;
```

在结构体中定义的常数可以在该结构体中随意使用；在进程中定义的常数只限该进程中使用。

在 VHDL 语言中信号、变量和常数都可以具有 VHDL 预定义的数据类型，或者也可以具有用户自定义的数据类型。

2.17　数组

数字系统经常使用存储数组。VHDL 数组可以用来指定储存这些数组的值。VLSI 电路的一个重要特点是重复使用类似的结构进。VHDL 数组可以用于模拟这种重复。

为了在 VHDL 中使用数组，必须先定义数组类型和数组对象。下例中的说明语句定义了一个名为 SHORT_WORD 的 1 维数组：

```
type SHORT_WORD is array (15 downto 0) of bit
```

这个数组有一个整数下标，取值范围为 `15 downto 0`，且数组中每个元素的数据类型均为 bit。这个新建的数据类型的名字为 `SHORT_WORD`。我们注意到，其实 `SHORT_WORD` 就是一个长度为 16 的 bit_vector。

这样，我们可以定义具有 SHORT_WORD 数据类型的数组对象：

```
signal DATA_WORD: SHORT_WORD;
variable ALT_WORD: SHORT_WORD : = "0101010101010101";
constant ONE_WORD: SHORT_WORD : = (others => '1');
```

以上语句定义了三个不同的数组。*DATA_WORD* 是一个 16 位的信号数组，下标从 15 到 0，每一位的初值为默认值‘0’；*ALT_WORD* 是 16 位变量数组，初始值为 0、1 交替的字符串；*ONE_WORD* 是 16 位常数数组，它所有位被（`others => '1'`）置为 1。

通过指定下标我们可以访问数组中的单个元素。例如，*ALT_WORD*(0) 表示 *ALT_WORD* 数组

的最右边一位；通过给出下标值的范围，也可以指定数组的一部分，*ALT_WORD*(5 **downto** 0)就表示 *ALT_WORD* 数组的低 6 位，其初值为“010101”。

数组类型和数组对象的定义书写格式为

```
type array_type_name is array index_range of element_type;
signal array_name: array_type_name [ : = initial_values ];
```

以上说明中，**signal** 可以是 **variable** 或 **constant**。

2.17.1 矩阵

数组也可以是二维或多维的。下例定义了一个二维数组变量，它表示一个 4 行 3 列的整数矩阵。

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;
variable matrixA: matrix4x3 : = ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));
```

变量 matrixA 将初始化为

$$
\begin{matrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12
\end{matrix}
$$

数组元素 matrixA(3, 2)表示矩阵中第 3 行、第 2 列的元素，取值为 8。

在定义数组类型时，若没有给出数组大小，那么该数组称为无约束数组类型，例如，

```
type intvec is array (natural range <>) of integer;
```

说明了 intvec 是一个无约束一维整数数组，它的大小为全体自然数。数组下标的默认类型为整数，但是也可以指定其他类型。由于无约束数组类型的下标范围没有限定，因此数组对象说明中必须给出下标的范围。例如，

```
signal intvec5: intvec(1 to 5) : = (3, 2, 6, 8, 1);
```

定义了一个名为 intvec5 的信号数组,下标范围为 1～5，初值为（3, 2, 6, 8, 1）。下面的说明语句定义了一个二维数组，没有给定行和列的下标范围。

```
type matrix is array (natural range <>, natural range <>) of integer;
```

例 1 在数字通信中我们经常使用奇偶校验位以便进行错误检测和纠正。最简单的方式就是在要传输的数据后再加上一位奇偶校验位。请用 VHDL 数组表示一个 5 位奇校验生成器，其中 4 位输入数据，要使用查表（LUT）方法。

解： 输入数据字为 4 位二进制数。一个 5 位奇校验表示输出字中 1 的个数为奇数。这个可以采用查表法，使用一个大小为16入口×5比特 的只读存储器（ROM）来实现。表格内容如图 2.64 所示。

输入（LUT 地址）				输出（LUT 数据）				
A	B	C	D	P	Q	R	S	T
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0	0
0	0	1	1	0	0	1	1	1
0	1	0	0	0	1	0	0	0
0	1	0	1	0	1	0	1	1
0	1	1	0	0	1	1	0	1
0	1	1	1	0	1	1	1	0
1	0	0	0	1	0	0	0	0
1	0	0	1	1	0	0	1	1
1	0	1	0	1	0	1		1
1	0	1	1	1	0	1	1	0
1	1	0	0	1	1	0	0	1
1	1	0	1	1	1	0	1	0
1	1	1	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1

图 2.64　一个校验码生成器的 LUT

该奇偶校验生成器的 VHDL 代码如图 2.65 所示。这里，我们使用 IEEE numeric_bit 包集合，*X* 和 *Y* 定义为无符号矢量，输出的头 4 位等于输入。因此，我们不必把所有 5 位输出都储存，而只需储存奇偶校验位，然后把它拼接到输入数据上即可。在图 2.65 的程序代码中，我们定义了一个新的 16 位数组类型 OutTable，并通过以下语句把 OutTable 定义为常数数组：

```
type OutTable is array (0 to 15) of bit;
```

该数组的下标为整数，其取值范围为 0～15。因此，无符号矢量 *X* 需要首先转换为整数。使用库中定义的 to_integer 函数就可以完成这种转换。

```
library IEEE;
use IEEE.numeric_bit.all;

entity parity_gen is
  port(X: in unsigned(3 downto 0);
       Y: out unsigned(4 downto 0));
end parity_gen;

architecture Table of parity_gen is
type OutTable is array(0 to 15) of bit;
signal ParityBit: bit;
constant OT: OutTable : = ('1', '0', '0', '1', '0', '1', '1', '0',
                           '0', '1', '1', '0', '1', '0', '0', '1');
begin
  ParityBit <= OT(to_integer(X));
  Y <= X & ParityBit;
end Table;
```

图 2.65　使用 LUT 法的校验码生成器

VHDL 中预定义的无约束数组类型有位矢量 bit_vector 型和字符串 string 型，其定义为

```
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;
```

字符串必须用双引号“ ”括起来。例如,“ This is a string ”表示一个字符串。下面的例子定义了一个常数 string1 字符串：

```
constant string1: string(1 to 29) : = "This string is 29 characters."
```

位矢量(bit_vector)可以写成字符串形式或者一系列用逗号分隔的位。例如，(‘1’, ‘0’, ‘1’, ‘1’, ‘0’)和 “10110”表示同一个位矢量。下例定义了一个常数位矢量 A，其下标的取值范围为 0~5，且初始值为“101011” ：

```
constant A : bit_vector(0 to 5) : = "101011";
```

当一个数据类型定义说明后，我们可以与此相关的子类型，以包含该类型下的一个子集。例如，在本节开始时定义的 SHORT_WORD 数据类型，可以定义为位矢量的一个子类型：

```
subtype SHORT_WORD is bit_vector (15 downto 0);
```

预定义的整数类型的两个子类型分别为 POSITIVE（包括所有正整数）和 NATURAL（包含正整数和 0)。

2.18 VHDL 中的循环语句

我们常常会遇到一些系统中的某些操作需要反复执行。VHDL 语言中的循环语句就可用于表示这种行为。循环语句是顺序语句。VHDL 有几种不同类型的循环语句，如 **for** 循环语句和 **while** 循环语句。

1. 无限循环

在一般计算机语言中不希望出现无限循环，但是在硬件模拟中无限循环却很有用，尤其是当一个设备连续工作，直到电源断开。

无限循环语句的一般格式为

```
[循环标号:] loop
  顺序语句
end loop [循环标号];
```

exit 语句的格式为

```
exit; 或 exit when 条件;
```

循环语句中可以包含 **exit** 语句。当循环执行到 **exit** 语句时，如果条件为真，则就无条件地从循环中跳出。

2. for 循环语句

如果想多次引用基本循环单元，则使用 **for** 循环语句，其中可以明确指定调用次数。**for** 循环语句的一般格式为

```
[循环标号:] for 循环变量 in 范围  loop
  顺序语句
end loop [循环标号];
```

当循环语句开始执行时，循环变量自动加载，不必另外定义，初始值为范围的第一个值，随后执行顺序语句。对循环变量的取值范围要设定，比如取值范围为 0～*n*，其中 *n* 可以是常数或变量。循环变量可以在循环体内的顺序语句中使用，但是在循环体内不能变化。一个循环执行完毕后，循环变量才被赋予范围内的第二个值，依此类推。当范围内的所有值都被遍历后，循环结束。当循环结束后，循环变量就不再用了。

我们可以在行为描述中使用这种循环语句。我们从 4 位全加器中摘取了部分语句。当 **for** 循环语句开始执行时，循环变量（i）初始化为 0，开始执行顺序语句，且当 $i=1, i=2$ 和 $i=3$ 时，顺序语句重复执行；然后循环结束。在循环结束前，一次迭代中的进位输出（*cout*）赋值给下一次迭代用的进位输入（*cin*）。由于和（*sum*）与进位（*carry*）的都是变量，所以进位输出的更新在瞬间完成。像这样的循环语句经常出现在 VHDL 的函数和过程语句中（将在第 8 章中介绍）：

```
loop1: for i in 0 to 3 loop
  Cout : = (A(i) and B(i)) or (A(i) and cin) or (B(i) and cin) ;
  sum(i) : = A(i) xor B(i) xor cin;
  cin : = cout;
end loop loop1;
```

我们还可以用循环语句生成一个基本单元的多个复制单元。当前面的代码进行综合时，综合器基本上会提供 4 个 1 位行波进位加法器。

3．while 循环语句

在 **for** 循环语句中，循环变量不能被程序所改变。但是，在 **while** 循环语句中循环变量可以被程序操作。因此在 **while** 循环语句中，可以使循环变量加 2。像大多数语言一样，**while** 循环语句迭代之前先检测条件是否满足，如果条件不满足，则循环终止。**while** 循环语句的一般格式为

```
[循环标号:] while 条件 loop
  顺序语句
end loop [循环标号];
```

这种构造一开始就是为了仿真。

图 2.66 是一个递减计数器的 VHDL 程序代码。我们使用 **while** 循环语句实现连续减操作，直到计数器变为 0 时循环结束。在每个时钟上升沿计数器都减 1，直到 count 为 0 或 stop 为 1 时，循环结束。

```
while stop = '0' and count / = 0 loop
  wait until clk'event and clk = '1';
    count <= count - 1;
  wait for 0 ns;
end loop;
```

图 2.66　while 循环的使用

2.19 Assert 和 Report 语句

当一个系统的 VHDL 模型建立完之后，下一步就是对其进行测试。一个 VHDL 模块必须经过测试和验证后才能很好地使用。VHDL 语言提供了一些特殊语句，如 **assert**（断言），**report**（报告）和 **severity**（严重程度）语句，用于辅助测试和验证，。

assert 语句用于验证某个条件是否为真，如果不是，则显示错误消息。assert 语句的一种格式为

```
assert 布尔表达式
   report 字符串表达式
   [severity  严重级别;]
```

assert 语句设定一个布尔表达式用来表示应满足的条件。如果条件不满足，则产生违规断言。如果在仿真过程中出现违规断言，则仿真器把该语句中的字符串表达式用 **report** 语句报告出来。如果布尔表达式为假，则字符串表达式及其严重级别将被显示在监视器上；如果布尔表达式为真，则不显示任何信息。严重级别有 4 级：注意、警告、错误和失败（即 note，warning，error 和 failure）。我们使用它们来反映某一违规的断言对该模块操作的影响程度。例如，严重的违规标记为故障，较小的违规标记为注意或警告。不同的仿真器其对这些违规严重级别的处理也不同。

如果 **assert** 语句被省略，则总报告消息。这样，语句

```
report "ALL IS WELL";
```

每当执行的时候，监视器上总是显示消息“ALL IS WELL”。

Assert 和 report 语句在创建测试平台（test benches）时非常有用。测试平台就是一个 VHDL 程序，它通过提供各种输入组合对 VHDL 编写的系统进行测试。它给待测系统或电路提供激励。在仿真中，测试平台被频繁用来为测试中的 VHDL 模块或电路提供一系列输入序列。图 2.67 给出了一个测试平台，用于测试在本章中较早建立的 4 位二进制加法器模块。我们把待测加法器作为一个元件嵌入到该测试平台程序中。测试平台生成的信号与加法器直接的接口如图 2.67 所示。在图 2.68 的测试平台代码中，我们使用常数数组表示加法器测试输入和期望输出，并使用 **for** 循环语句从数组中取出输入数据；使用 **assert** 和 **report** 语句检查输出，然后报告对于特定的输入组合其输出是否与期望值相符。**assert** 语句只有对仿真才有意义。在综合时，综合器可以简单地忽略其存在，就当不存在。

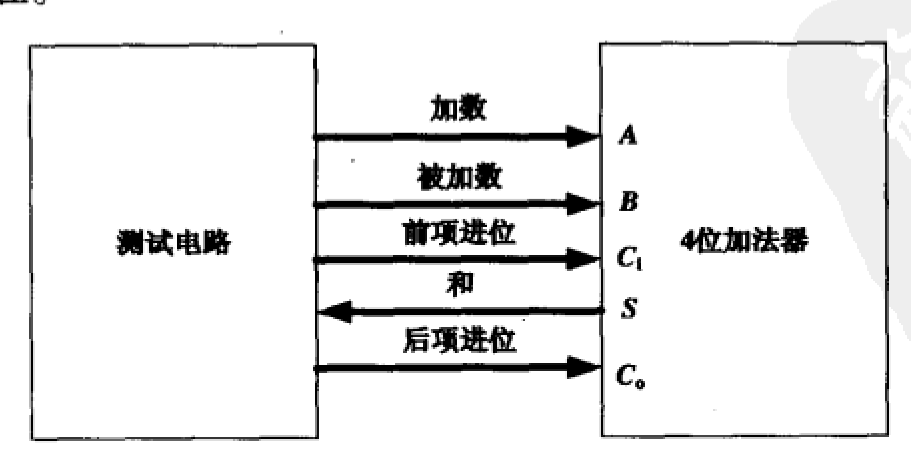


图 2.67　当使用测试程序对 4 位加法器进行测试时使用的交互信号

```
entity TestAdder is
end TestAdder;

architecture test1 of TestAdder is
component Adder4
   port(A, B: in bit_vector ( 3 downto 0 ); Ci: in bit;
         S: out bit_vector (3 downto 0); Co: out bit);
 end component;
 constant N: integer : = 11;
 type bv_arr is array (1 to N) of bit_vector( 3 downto 0 );
type bit_arr is array (1 to N) of bit;
constant addend_array: bv_arr : = ("0111", "1101", "0101", "1101",
          "0111", "1000", "0111", "1000", "0000", "1111", "0000");
constant augend_array: bv_arr : = ("0101", "0101", "1101", "1101",
          "0111", "0111", "1000", "1000", "1101", "1111", "0000");
constant cin_array: bit_arr : = ('0', '0', '0', '0', '1', '0', '0',
          '0', '1', '1', '0', );
constant sum_array: bv_arr : = ("1100", "0010", "0010", "1010",
          "1111", "1111", "1111", "0000", "1110", "1111", "0000");
constant cin_array: bit_arr : = ('0', '1', '1', '1', '0', '0', '0',
          '1', '0', '1', '0', );
signal addend, augend, sum: bit_vector ( 3 downto 0 );
signal cin, cout: bit;
begin
   process
   begin
     for i in 1 to N loop
        addend <= addend_array(i);
        augend <= augend_array(i);
        cin <= cin_array(i);
        wait for 40 ns;
        assert (sum = sum_array(i) and cout = cout_array(i))
           report "Wrong Answer"
           severity error;
      end loop;
      report "Test Finished";
.  end process;
   add1: adder4 port map (addend, augend, cin, sum, cout);
end test1;
```

图 2.68　4 位加法器的检测程序

下面，我们再举一个例子来说明在测试平台中怎样提供波形输入。在本章的较早例子中，我们使用仿真器命令对 VHDL 模块进行了测试。图 2.69 是一个 VHDL 测试程序，它所做的测试与图 2.55 中使用仿真器命令所做的测试完全一样。我们用下面的语句生成一个时变信号，输入给 X：

```
X <= '0' , '1' after 350 ns , '0' after 550 ns, '1' after 750 ns, '0'
       after 950 ns, '1' after 1350 ns;

entity test_code_conv is
end test_code_conv;

architecture tester of test_code_conv is
signal X, CLK, Z: bit;
component Code_Converter is
  port(X, CLK: in bit;
       Z: out bit);
end component;
begin
  clk <= not clk after 100 ns;
  X <= '0', '1' after 350 ns, '0' after 550 ns, '1' after
       750 ns, '0' after 950 ns, '1' after 1350 ns;
  CC: Code_Converter port map (X, clk, Z);
end tester;
```

图 2.69 生成码转换器检测程序的检测序列

本章中，我们涉及了 VHDL 的基本内容。我们介绍了如何用 VHDL 模拟组合逻辑电路和时序逻辑电路。由于 VHDL 是一种硬件描述语言，所以它在几个方面与一般的编程语言不同。最重要的是 VHDL 语句是并发执行的，这是因为它必须模拟实际硬件电路，该电路的每个元件的所有操作都是同时执行的。进程中的语句是顺序执行的，但是每个进程本身都是并行执行的。VHDL 信号是模拟硬件中的实际信号，但是变量可以用于内部计算，它局部存在于进程、过程和函数之中。VHDL 的高级特性将在第 8 章中介绍。

习题

2.1 (a) VHDL 和 VHSIC 分别是什么意思？

(b) 硬件描述语言（如 VHDL）与普通的编程语言有什么不同？

(c) 在设计中用硬件描述语言比图形输入有何好处？

2.2 (a) 下列符号中哪些是合法的 VHDL 标识符？`123A`, `A_123`, `_A123`, `A123_`, `c1_c2`, `and` 和 `and1`

(b) 下列哪些标识符是等效的？`aBC`, `ABC`, `Abc`, `abc`

2.3 设有下面的 VHDL 并发语句：

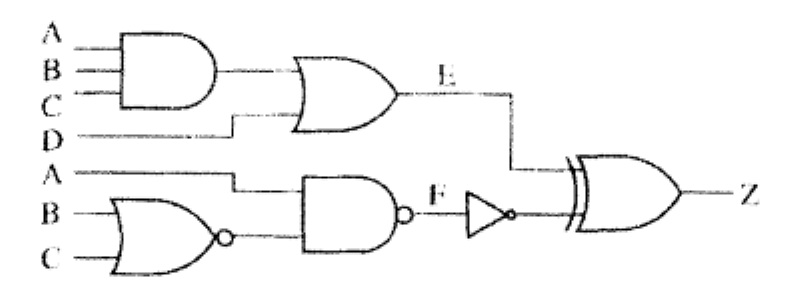
B <= A **and** C **after** 3 ns;

C <= **not** B **after** 2 ns;

(a) 画出该语句表示的电路图。

(b) 画出电路的时序图。初始化 $A = B =$ '0', $C =$ '1'，并且 A 在 5 ns 时变为 '1'。

2.4 用 VHDL 并行语句描述下面的组合逻辑电路。每个门有 5 ns 的延时（不包括反相器），反相器的延时为 2 ns。



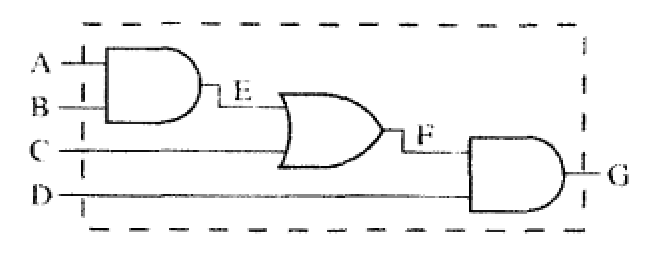
2.5 (a) 利用逻辑表达式写出全减器的 VHDL 代码。

(b) 把(a)设计的全减器作为元件，写出一个 4 位全减器的 VHDL 代码。

2.6 写出下面电路的 VHDL 代码。假设可以忽略门延迟。

(a) 使用并行语句。

(b) 使用带有顺序语句的进程。



2.7 在下面的 VHDL 代码中，*A*, *B*, *C* 和 *D* 均为整数，且在 10 ns 时赋为 0。如果在 20 ns 时 *D* 从 0 变为 1，试确定 *A*, *B* 和 *C* 发生变化的时间和取值。

```
process(D)
begin
  A <= 1 after 5 ns;
  B <= A + 1;              -- 在A改变前执行
  C <= B after 10 ns;      -- 在B改变前执行
end process;
```

2.8 (a) 下面的 VHDL 代码表示什么电路？

```
process(CLK, Clr, Set)
begin
  if Clr = '1' then Q<= '0';
  elsif Set = '1' then Q<= '1';
  elsif CLK'event and CLK <= '0' then
   Q<= D;
 end if;
end process;
```

(b) 如果 *Clr* = *Set* = ‘1’，则（a）中的电路会怎样？

2.9 写出 SR 锁存器的 VHDL 代码（使用一个进程）。

2.10 一个 M-N 触发器对时钟下降沿的响应如下：

如果 *M* = *N* = ‘0’，则触发器改变状态。

如果 *M* = ‘0’且 *N* = ‘1’，则触发器输出置‘1’。

如果 *M* = ‘1’ 且 *N* = ‘0’，则触发器输出置‘0’。

如果 *M* = *N* = ‘1’，则触发器状态不改变。

当 *CLRn* = ‘0’时，触发器异步清零。

写出实现该 M-N 触发器的完整 VHDL 代码。



2.11 DD 触发器与 D 触发器类似（$Q^+=D$），只是它在时钟的上升沿和下降沿均发生状态改变。它有一个直接复位输入 R，当 R = '0'时，不论时钟如何，触发器的输出 Q = '0'。同样，它还有一个置位输入 S，无论时钟如何，触发器的输出 Q = '1'。写出 DD 触发器的 VHDL 程代码。

2.12 一个抑制翻转的触发器有输入 *I0*, *I1*, *T* 和 *Reset* 及输出 *Q* 和 *QN*。*Reset* 为高电平有效的置零端，其优先级最高。该触发器工作过程如下：当 *I0* = '1'时触发器在 *T* 上升沿改变状态，当 *I1* = '1'时在 *T* 下降沿改变状态，当 *I0* = *I1* = '0'时不发生状态改变（*Reset* 端无效时）。若从 *T* 到输出的传输延时为 8 ns，从 *Reset* 到输出的传输延时为 5 ns。

(a) 写出该触发器的完整 VHDL 代码。

(b) 写出仿真器命令，用于测试该触发器对下面输入序列的响应。输入序列为：*I1* = '1'，*T* 翻转 2 次，*I1* ='0'，*I0* = '1'时 *T* 翻转 2 次。

2.13 在下面的 VHDL 进程语句中，*A*, *B*, *C* 和 *D* 均为整数，且在 10 ns 时赋为 0。如果在 20 ns 时 *E* 从'0'变为'1'，试给出每个信号变化的时间和取值，并按时间顺序写出这些变化（20, 20+Δ, 20+2Δ, …）。

```
p1: process
begin
 wait on E;
  A <= 1 after 5 ns;
  B <= A+1;
  C <= B after 10 ns;
  wait for 0 ns ;
  D <= B after 3 ns;
  A <= A +5 after 15 ns;
  B <= B +7;
end process p1;
```

2.14 在下面的 VHDL 进程语句中，*A*, *B*, *C* 和 *D* 均为整数，且在 10 ns 时赋为 0。如果在 20 ns 时 *E* 从'0'变为'1'，试给出每个信号的变化时间和取值。并按时间顺序写出这些变化（20, 20+Δ, 20+2Δ, …）。

```
p2: process (E)
begin
  A <= 1 after 5 ns;
  B <= A+1;
  C <= B after 10 ns;

  D <= B after 3 ns;
  A <= A +5 after 15 ns;
  B <= B +7;
end process p2;
```

2.15 在下面的 VHDL 代码中，如果 *D* 在 5 ns 时变为 1，试给出 *A*, *B*, *C*, *D*, *E* 和 *F* 的每个变化取值，即给出在 5, 5+Δ, 5+2Δ, …等时刻的取值。直至下列三个条件中的任意一个得到满足：进行到 20 步；不再发生改变；信号取值出现重复。

```
entity prob is
```

```
  port (D: inout bit );
end prob;

architecture q1 of prob is
  signal A, B, C, E, F: bit;
begin
  C <= A;
  A <= (B and not E) or D;
  P1: process (A)
  begin
    B <= A;
  end process P1;
  P2: process
  begin
    wait until A = '1';
    wait for 0 ns;
    E <= B after 5 ns;
    D <= '0';
    F <= E ;
  end process P2;
end architecture q1;
```

2.16 假设 *B* 由下列仿真命令驱动:

```
force B 0 0, 1 10, 0 15, 1 20, 0 30, 1 35
```

当下列并发语句执行时，画出描述 *A*, *B* 和 *C* 的时序图:

```
A <= transport B after 5 ns;
C <= B after 8 ns;
```

2.17 假设 *B* 由下列仿真命令驱动:

```
force B 0 0, 1 4, 0 10, 1 15, 0 20, 1 30, 0 40
```

当下列并发语句执行时，画出描述 *A*, *B* 和 *C* 的时序图。

```
A <= transport B after 5 ns;
C <= B after 5 ns;
```

2.18 在下面的 VHDL 进程语句中, *A*, *B*, *C* 和 *D* 均为信号且数据类型为 bit，且在 4 ns 时赋为'0'。在 5 ns 时 *A* 从 0 变为 1，列表写出不同时刻 *A*, *B*, *C* 和 *D* 的值（时间取到 18 ns），并指出每个进程开始执行的时间（Δ时间也考虑在内）。

```
p1: process(A)
begin
  B <= A after 5 ns;
  C <= B after 2 ns;
end process;
p2: process
begin
```

```
    wait on B;
    A <= not B;
    D <= not A xor B;
  end process;
```

2.19 如果 A = “101”, B = “011”, C = “010”，则下列语句的取值为多少?

(a) (A & B) **or** (B & C)

(b) A **ror** 2

(c) A **sla** 2

(d) A & **not** B = “111110”

(e) A **or** B **and** C

2.20 考虑下面的 VHDL 代码:

```
entity Q3 is
  port (A, B, C, F, Clk: in bit ;
         E: out bit );
end Q3;

architecture Qint of Q3 is
  signal D, G: bit;
begin
  process (Clk)
  begin
    if Clk'event and Clk = '1' then
      D <= A and B and C;
      G <= not A and not B;
      E <= D or G or F;
    end if ;
   end process;
 end Qint;
```

(a) 画出该电路的框图（不用逻辑门，只画框图）。

(b) 给出上面代码实现的电路图（画逻辑门）。

2.21 为了实现下面的 VHDL 代码，我们使用的硬件为：多个带时钟使能的 D 触发器、一个多路选择器、一个加法器和任意逻辑门。设 *Ad* 和 *Ora* 不能同时为 1，且只要其中的一个为 1 时，触发器才开始工作。

```
library IEEE;
use IEEE.numeric_bit.all;

entity module1 is
port(A, B: in unsigned (2 downto 0 );
     Ad, Ora, clk: in bit;
     C: out unsigned (2 downto 0));
end module1;

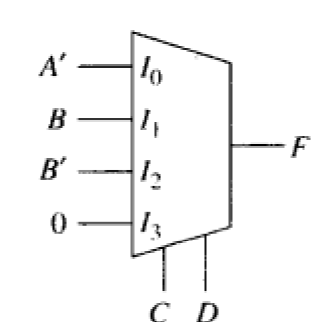
architecture RT of module1 is
```

```
begin
  process ( clk )
  begin
    if clk = '1' and clk'event then
      if Ad = '1' then C <= A + B; end if ;
      if Ora = '1' then C <= A or B; end if ;
    end if;
  end process;
end RT;
```

2.22 试画出下列 VHDL 进程描述的电路。只使用两个门。

```
process ( clk , clr )
begin
  if clr = '1' then Q <= '0';
  elsif clk'event and clk = '0' and CE = '1' then
     if C = '0' then Q <= A and B;
     else Q <= A or B; end if ;
  end if;
end process;
```

2.23 (a) 写出可以描述下面 4 选 1 MUX 的选择信号赋值语句。假设 MUX 存在 10 ns 的固有延迟，即在输入改变后，经过 10 ns 延迟输出才发生改变。



(b) 用条件信号赋值语句重做(a)题

(c) 用带有 case 语句的进程重做(a)题。

2.24 (a) 写出与下列并行语句等价的 VHDL 进程：

```
A <= B1 when C = 1 else B2 when C = 2 else B3 when C =
     3 else 0;
```

(b) 画出下列 VHDL 语句所描述的电路图：

```
A <= B1 when C1 = '1' else B2 when C2 = '1' else B3 when C3 = '1' else 0;
```

2.25 写出 SR 锁存器的 VHDL 代码。

(a) 使用条件赋值语句。

(b) 使用特征多项式。

(c) 使用两个逻辑门。

2.26 观察如图 2.38 所示的 VHDL 代码，如果 *A* = “1101”, *B* = “111”和 *Ci* = ‘1’，则 *S* 和 *Co* 的值为多少?

2.27 一个正整数 *B* (*B* <16)与一个 4 位的位矢量 *A* 求和，得到一个 5 位的位矢量。写出实现该计算的 VHDL 代码。使用 IEEE numeric bit 包集合中的重载运算符实现加操作，转换功能可以通过函数调用来实现，最终结果为一个位矢量，而不是一个无符号矢量。

2.28 现有一个 4 位数的幅值比较器芯片（如 74LS85），它可以对两个 4 位数 *A* 和 *B* 进行比较，有三种输出：*A*<*B*, *A* = *B* 或 *A*>*B*。现有三个输出信号代表上边的三种比较结果。注意，在任何时刻输出线中只有一条为高电平，其余两条均为低电平。该芯片是可以级联的，有三个

输入：$A>B.IN, A=B.IN$ 和 $A<B.IN$，这样可以级联该芯片构成 8 位或更多位幅值比较器。

(a) 画出此 4 位数幅值比较器的框图。

(b) 用两个 4 位数幅值比较器构成一个 8 位幅值比较器，并画出实现框图。

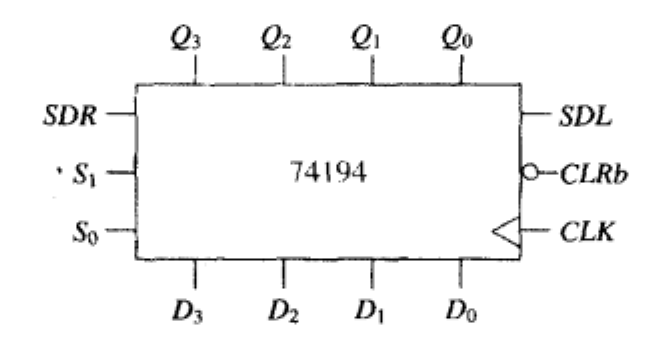
(c) 写出 4 位数比较器的行为描述 VHDL 代码。

(d) 写出 8 位数比较器的 VHDL 代码，其中把 4 位数比较器作为组成元件。

2.29 写出 16 位串入、串出的移位寄存器的 VHDL 代码。该寄存器具有以下端口：*SI*（串行输入），*EN*（使能端），*CK*（时钟，上升沿有效）和 *SO*（串行输出）。

2.30 74194 4 位双向移位寄存器，其工作过程如下：

CLRb 为异步输入端，低电平有效。寄存器在时钟上升沿时发生状态改变。当控制端 $S_1=S_0=1$ 时，寄存器并行置位；当 $S_1=1, S_0=0$ 时寄存器向右移位，并把 *SDR* 的值赋给 Q_3；当控制端 $S_1=S_0=1$ 时，寄存器并行置位（读入 $D_3D_2D_1D_0$）。当 $S_1=0, S_0=1$ 时寄存器向左移位，并把 *SDL* 的值赋给 Q_0；当 $S_1=S_0=0$ 时不进行任何操作。



(a) 用行为描述方式写出 74194 的 VHDL 模块。

(b) 画出由 2 个 74194 构成的 8 位双向移位寄存器的框图，并编写其 VHDL 代码（用 74194 作为元件）。该 8 位寄存器的并行输入、输出端分别为 X(7 downto 0)和 Y(7 downto 0),而串行输 入端为 RSD 和 LSD。

2.31 输出为 *Q* 的 4 位递增/递减同步十进制计数器的工作过程如下：在 *CLK* 输入上升沿时状态发生改变。它其具有异步清 0 端 *CLR*。当 $CLR=0$ 时，系统强制清 0。

如果输入 $LOAD=0$，数据输入端 D 置入到计算器。

如果 $LOAD=ENT=ENP=UP=1$，则计数器开始递增计数。

如果 $LOAD=ENT=ENP=1$，则计数器递减计数。

如果 $ENP=UP=1$ 且在状态 9，则进位输出端(CO)=1。

如果 $ENP=1$，$UP=0$ 且在状态 0，则进位输出端(CO)=1。

(a) 写出该计数器的 VHDL 代码。

(b) 把两个 4 位递增/递减同步十进制计数器级联构成一个十进制计数器，此计数器可以从 00 向 上数到 99，也可从 99 向下数到 00。写出该计数器的 VHDL 代码并画出实现框图。

(c) 按下列顺序进行仿真：计数器置数为 98，递减计数 3 次，两个时钟内不做任何操作，再递减计数 4 次，然后清零。

2.32 写出 74HC192 同步 4 位递增/递减计数器的 VHDL 模块。忽略所有时间数据。你的代码必须包含语句 **process** (DOWN, UP, CLR, LOADB)。

2.33 考虑下面的 8 位双向同步移位寄存器。它可以并行置入，其输入、输出管脚的符号如下：

CLR 异步清零，覆盖所有其他输入

Q(7:0) 8 位输出

$D(7{:}0)$　8 位输入
$S0$，$S1$　模式控制输入
LSI　左移串行输入端
RSI　右移串行输入端

模式控制输入工作情况如下：

S0	S1	操作
0	0	无操作
0	1	右移
1	0	左移
1	1	并行载入数据（如 $Q = D$）

(a) 写出该移位寄存器 VHDL 代码的实体部分。
(b) 写出该移位寄存器 VHDL 代码的结构体部分。
(c) 两个 8 位双向同步移位寄存器可以组成一个 16 位循环移位寄存器。此 16 位寄存器由信号 L 和 R 控制。当 L = 1，R = 0 时寄存器循环左移；当 L = 0，R = 1 时寄存器循环右移；当 L = R = 1 时寄存器从 X(15:0)置位；如果 L = R = 0，则寄存器无任何操作。画出该 16 寄存器的实现框图。
(d) 写出 16 位移位寄存器 VHDL 代码的实体部分
(e) 写出 16 位移位寄存器 VHDL 代码的结构体部分，并使用(a)和(b)中的模块。

2.34 完成下面的 VHDL 代码，以实现了一个计数器，对下面的顺序计数：Q = 1000，0111，0110，0101，0100，0011，1000，0111，0110，0101，0100，00110，…（重复）。当 *Ld8* = ‘1’时，计数器同步置入 1000。当 *Enable* = ‘1’时，计数器按照上面的计数顺序操作。无论何时只要在状态 0101，计数器就输出 *S5* = ‘1’。不要对实体做任何改变，而且你所编写的代码必须能够综合。

```
library IEEE;
use IEEE.numeric_bit.all;

entity countQ1 is
  port (clk, Ld8, Enable: in bit; S5: out bit;
       Q: out unsigned (3 downto 0));
  end countQ1;
```

2.35 一个同步 4 位递增/递减二进制计数器具有一个同步清零信号 *CLR* 和一个同步置入信号 *LD*。*CLR* 的优先级高于 *LD*，且二者均为高电平有效。*D* 为计数器的 4 位输入，*Q* 为 4 位输出。*UP* 为计数方向控制信号。如果 *CLR* 和 *LD* 均未被激活，且 *UP* = 1，则计数器向上计数。如果 *CLR* 和 *LD* 均未被激活，且 *UP* = 0，则计数器向下计数。所有改变均在时钟下降沿进行。
(a) 写出该计数器的行为描述 VHDL 代码。
(b) 使用上面的递增/递减计数器实现一个模 6 同步计数器。要求模 6 计数器可以从 1 数到 6，有 一个外部复位端，当有效时，计数器置 1。计数使能信号 *CNT* 使计数顺序为 1, 2, 3, 4, 5, 6, 1, 2, …每个时钟脉冲加 1。你可以使用任何逻辑使计数器从 6 变回为 1。该模 6 计数器只能向上计数。给出该计数器的文字和框图描述。
(c) 写出(b)中模 6 计数器的行为描述 VHDL 代码。

2.36 检查下列 VHDL 代码并完成后面的练习。

```
entity Problem
  port (X, CLK : in bit;
        Z1, Z2: out bit);
end Problem;

architecture Table of Problem is
  signal State, Nextstate: integer range 0 to 3 : = 0;
begin
 process (State, X)    -- 组合电路
 begin
  case State is
    when 0 = >
      if X = '0' then Z1<= '1'; Z2 <= '0'; Nextstate <= 0;
      else Z1<= '0'; Z2 <= '1'; Nextstate <= 1; end if;
    when 1 = >
      if X = '0' then Z1<= '0'; Z2 <= '1'; Nextstate <= 1;
      else Z1<= '0'; Z2 <= '1'; Nextstate <= 2; end if;
    when 2 = >
      if X = '0' then Z1<= '0'; Z2 <= '1'; Nextstate <= 2;
      else Z1<= '0'; Z2 <= '1'; Nextstate <= 3; end if;
    when 3 = >
      if X = '0' then Z1<= '0'; Z2 <= '0'; Nextstate <= 0;
      else Z1<= '1'; Z2 <= '0'; Nextstate <= 1; end if;
   end case;
  end process;
  process (CLK)            --State Register
  begin
     if CLK'event and CLK = '1' then  -- 时钟上升沿
       State <= Nextstate;
    end if;
  end process;
end Table;
```

(a) 画出该代码实现电路的框图。

(b) 写出该代码实现的状态表。

2.37 (a)试给出题 1.13 中设计的状态机的行为描述 VHDL 代码。假设在时钟脉冲下降沿时状态机发生 状态改变。不用 if-then-else 语句，而用数组表示状态表和输出表。编译和仿真你所编写的程序。测试序列为 X=1101 1110 1111(X 在时钟下降沿过后 1/4 时钟周期改变赋值)。

(b) 用数据流描述方式，并使用下一状态和输出的逻辑表达式编写(a)中状态机的 VHDL 代码，并通过仿真结果指出在何时 S 和 V 可以被读出。

(c) 用结构描述方式，并使用电路构造图（门电路和 J-K 触发器构成）编写(a)中状态机的 VHDL 代码。在这一部分，你可能会用到 BITLIB 库。

2.38 (a) 写出在题 1.14 中设计的状态机的行为描述 VHDL 代码。假设在时钟脉冲下降沿时状态机发生状态改变。使用 case 和 if-then-else 语句描述状态图，然后编译和仿真你所编写的程

序。测试序列为 $X = 1011\ 0111\ 1000$（X 在时钟下降沿过后 1/4 时钟周期改变赋值）。

(b) 用数据流描述方式，并使用下一状态和输出的逻辑表达式编写(a)中状态机的 VHDL 代码，并通过仿真结果指出在何时 D 和 B 可以被读出。

(c) 用结构描述方式，并使用电路构造图（门电路和 J-K 触发器构成）编写(a)中状态机的 VHDL 代码。在这一部分，你可能会用到 BITLIB 库。

2.39 一个 Moore 时序电路有 2 个输入（X_1 和 X_2）和 1 个输出（Z），其状态表如下：

当前状态	下一状态 $X_1X_2 = 00$	01	10	11	输出 (Z)
1	1	2	2	1	0
2	2	1	2	1	1

写出行为描述的 VHDL 代码。假设在时钟下降沿之后 10 ns 状态发生改变。在状态发生改变 10 ns 后输出发生改变。

2.40 写出实现下面状态表的 VHDL 代码。使用两个进程。状态改变均发生在时钟下降沿。使用并发条件语句实现 Z_1 和 Z_2。假设该时序电路的组合逻辑部分存在 10 ns 的传输延迟，从时钟上升沿到状态寄存器输出的传输延迟为 5 ns。

当前状态	下一状态 $X_1X_2 = 00$	01	11	输出 (Z_1Z_2)
1	3	2	1	00
2	2	1	3	10
3	1	2	3	01

2.41 在下面的代码中，*state* 和 *nextstate* 为取值范围时 0 ~ 2 的整数。

```
process ( state, X)
begin
   case state is
     when 0 = > if X = '1' then nextstate <= 1;
     when 1 = > if X = '0' then nextstate <= 2;
     when 2 = > if X = '1' then nextstate <= 0;
  end case;
end process;
```

(a) 解释为什么该程序综合时会生成一个锁存器？

(b) 在锁存器输出会出现什么信号？

(c) 优化程序，去掉锁存器。

2.42 在下面的进程中，A, B, C 和 D 均为整数，且在 10 ns 时值均为 0。若在 20 ns 时，E 从'0'变为'1'，试给出所有的变化。同时给出每个变化发生时刻、每个被影响的信号/变量和它们的取值。

```
process
  variable F: integer : = 1; variable A: integer : = 0;
begin
   wait on E;
   A : = 1;
   F : = A+5;
```

```
  B <= F+1 after 5 ns;
  C <= B+2 after 10 ns;
  D <= C +5 after 15 ns;
  A : = A+5;
end process;
```

2.43 下面的 4 选 1 MUX 程序模块有什么错误？（非语法错误）

```
architecture mux_behavioral of 4to1mux is
signal sel: integer range 0 to 3;
begin
  process (A, B, I0, I1, I2, I3)
  begin
    sel <= 0;
    if A = '1' then sel <= sel+1; end if;
    if B = '1' then sel <= sel+2; end if ;
    case sel is
      when 0 => F <= I0;
      when 1 => F <= I1;
      when 2 => F <= I2;
      when 3 => F <= I3;
    end case;
  end process;
end mux_behavioral;
```

2.44 在下面给出的 VHDL 代码进行仿真时，在 5 ns 时 *A* 变为'1'。在 40 ns 的仿真时间内，*A*, *B* 和 *D* 在每个时刻都有何变化？请列表说明。

```
entity Q1F00 is
   port (A: inout bit);
end Q1F00;

architecture Q1F00 of Q1F00 is
  signal B, D: bit;
begin
  D <= A xor B after 10 ns;
  process (D)
    variable C: bit;
  begin
    C : = not D;
    if C = '1' then
      A <= not A after 15 ns;
    end if;
    B <= D;
  end process;
end Q1F00;
```

2.45 下面的 VHDL 代码表示什么电路器件？

```
process (CLK, RST)
  variable Qtmp: bit;
begin
  if RST '1' then Qtmp : = '0';
  elsif CLK'event and CLK = '1' then
    if T = '1' then
      Qtmp : = not Qtmp;
    end if;
  end if;
  Q <= Qtmp;
end process;
```

2.46 (a) 写出 4 输入、3 输出 LUT 的 VHDL 代码。3 位输出为输入中含有 1 的个数（二进制表示）。

(b) 编写一个 VHDL 程序，它可以计算一个 12 位二进制数中 1 的个数。要求使用 3 个(a)中的模块，并使用重载操作符。

(c) 用下面的数据输入对你的程序进行仿真验证：

111111111111，010110101101，100001011100

2.47 用 LUT 实现一个 3-8 译码器。给出 LUT 的真值表，并写出 VHDL 代码。输入为 A, B 和 C，输出为 8 位无符号矢量。

2.48 A(1 to 20)是一个能够储存 20 个整数的数组。写出求数组中最大值的 VHDL 程序代码。

(a) 使用 for 循环。

(b) 使用 while 循环。

2.49 编写一个测试程序，对一个输入为 X，输出为 Z 的 Mealy 时序电路进行测试。程序必须把 Mealy 电路作为一个元件。假设 Mealy 电路在时钟上升沿改变，你所编写的程序必须生成一个周期为 100 ns 的时钟，并应用下面的检测序列：

$$X = 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0$$

X 在时钟 CLK 上升沿到来后 10 ns 发生改变。你所编写的代码必须在恰当的时间对 Z 进行读取，并验证是否生成下面的输出序列：

$$Z = 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0$$

如果 Mealy 电路的输出序列不正确，则报错；否则，报告“序列正确”。完成下面的测试器的结构体程序代码。

```
architecture test1 of tester is
  component Mealy
    -- 时序待测电路；假设该元件在你的设计中可用；不要给该元件编写代码
    port (X, CLK: in bit; Z: out bit);
end component;
signal XA: bit_vector (0 to 11) : = "011011011100";
signal ZA: bit_vector (0 to 11) : = "100110110110";
```

2.50 写出图 2.58 时序电路的的 VHDL 测试平台程序。你的测试平台要生成 10 个可能的输入序列（0000, 10000, 0100, 1100, …）并验证这些输入序列对应的输出序列是否正确。注意，元件均有 10 ns 的延迟。输出要在时钟上升沿后 1/4 时钟周期时发生改变，输出要在适当

的时刻进行读取。如果输出序列正确，则报告“通过”；否则，报告“失败”。

2.51 写出习题 2.34 中计数器的测试平台程序。你的测试平台必须生成一个周期为 100 ns 的时钟。在第一个时钟，计数器置数；在接下来的 5 个时钟内计数器计数；在接下来的两个时钟不工作；然后在持续工作 10 个时钟。无论何时只要 *S5* = ‘1’，则测试端口就输出当前时间（给出时间单位）。在测试平台程序中只允许使用并发语句。

2.52 完成下面的 VHDL 代码以实现测试平台，用于测试时序电路 SMQ1。假设时序电路 SMQ1 的 VHDL 代码可用。半个时钟周期为 50 ns。并应用输入序列 *X* = 1, 0, 0, 1, 1。假设该时序电路正确的输出序列应为 1, 1, 0, 1, 0。要求只使用一个并发语句生成 *X* 序列。你所编写的代码必须在恰当的时间，对输出 *Z* 进行读取，并与 *Z* 的正确值进行比较，并把正确答案储存在一个位矢量常数中：

answer（1 *to* 5） ="11010";

如果答案正确，则端口信号 *correct* 为 TRUE；否则为 FALSE。要求在正确的时刻读取 *Z*，而且在你的测试程序中要使用以下语句：

```
entity testSMQ1 is
 port (correct: out Boolean);
end testSMQ1;
architecture testSM of testSMQ1 is
 component SMQ1 -连续电路模块
   port (X, CLK: in bit; Z: out bit);
 end component;
 constant answer: bit_vector(1 to 5) : = "11010";
begin
```



第 3 章　可编程逻辑器件简介

在第 1 章中，我们介绍了如何使用不同的标准模块构建同一个数字电路。如果把这些模块都放到一个集成芯片上，而且给用户提供修改其配置的机制，那么我们就可以用这块芯片实现几乎所有的电路。这就是可编程逻辑器件的基本原理。

本章介绍如何使用可编程逻辑器件设计数字系统，包括只读存储器（ROM）、可编程逻辑阵列（PLA）和可编程阵列逻辑（PAL）器件，并进一步介绍复杂可编程逻辑器件（CPLD）和现场可编程门阵列（FPGA）。这些器件的使用使我们只用一个 IC 芯片就能应实现一个很复杂的逻辑函数，它需要很多门和触发器来实现。本章中我们只是简单地介绍 FPGA，更详细的讨论将在第 6 章中给出。

3.1　可编程逻辑器件简介

设计者总是喜欢用可编程逻辑器件（如 PAL 和 FPGA）进行数字电路设计，其原因主要有两点。首先，可编程逻辑器件有一定的集成度，在一个物理芯片上可以实现相当可观的许多函数功能。有了可编程逻辑器件，我们就可以不再使用很多的通用器件，也可以避免由于外部连线造成的许多不便和故障。其次，可编程逻辑器件的可重构能力日益增强。许多可编程逻辑器件都很容易重构编程。一般来说，根据出现的错误或指标的变化修改设计更容易的。当前的可编程逻辑器件主要分为两类：一类是一次性编程器件，一类是可重构编程多次的器件。

图 3.1 说明了常用可编程逻辑器件的分类。可编程逻辑可以分为两类：现场可编程逻辑和工厂可编程逻辑。所谓的“现场”是指那些在用户“现场”编程的器件，而不是在半导体生产厂家。许多人说的可编程逻辑器件，一般是指现场可编程器件，但是也有工厂可编程器件。这些逻辑器件是为了满足顾客的需要而在工厂编程的通用器件，所使用的编程技术是不可逆的，所以这些逻辑器件只能编程一次。掩模可编程门阵列（MPGA）和只读存储器（ROM）就属于工厂可编程逻辑。很多第一代的可编程逻辑器件都只能在工厂编程。

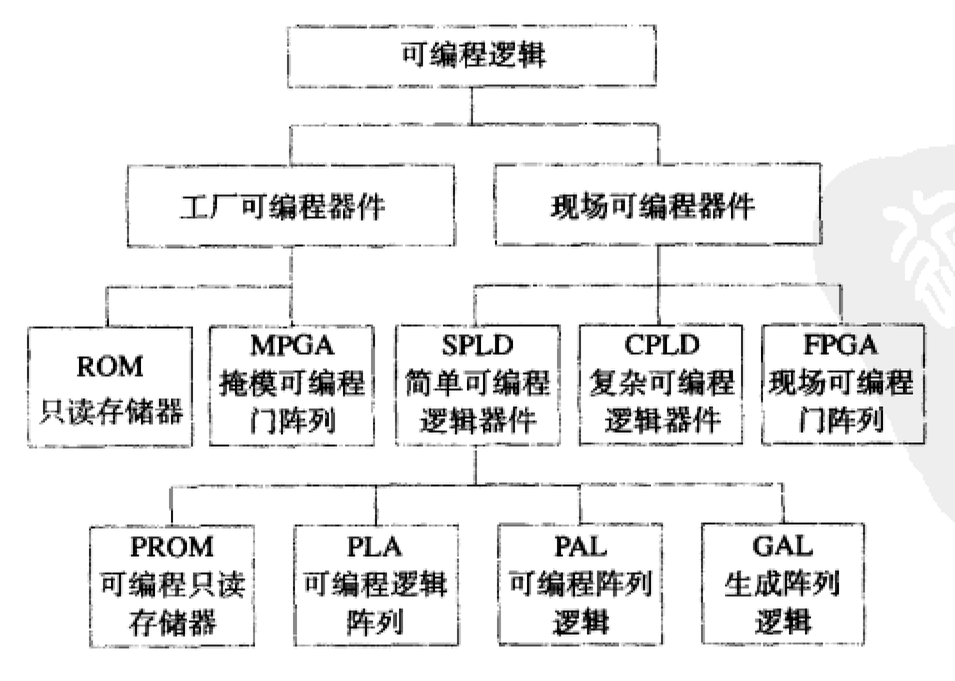


图 3.1　主要的可编程逻辑器件

只读存储器可以看做是早期的一种可编程逻辑器件。只读存储器虽然最初是作为存储器使用，但它也可以用来实现任何组合逻辑电路，我们将在 3.2.1 节介绍这一应用。MPGA 是一种传统门阵列，它需要设计一个掩模。通常 MPGA 简称为门阵列，是制造专用集成电路（ASIC）的常用技术。

20 世纪 70 年代初，人们开发了基于与或（AND-OR）电路的用户可编程逻辑。1972 年—1973 年，出现了允许设计者快速定制的一次性现场可编程逻辑阵列，一些人称这种器件为现场可编程逻辑阵列（FPLAs）。随后，MMI 公司（现已被 AMD 收购）生产出了一种集成电路称为可编程逻辑阵列（PLAs）。采用了 20×24 引线封装，它可以实现 5 ~ 20 个通用芯片的功能。一种类似器件是可编程阵列逻辑（PAL）。

PAL 和 PLA 中包含了很多的逻辑门。在 PLA 中既有可编程 AND 门阵列，也有可编程 OR 门阵列。这样用户就可以实现由这两种逻辑门组成的组合函数。PAL 是 PLA 的一种特例，在 PAL 中 OR 门阵列是固定的，只有 AND 门阵列可以编程。很多 PAL 中还含有触发器。

由于设计过程简单，在 20 世纪 70 年代和 80 年代 PAL 和 PLA 非常流行。MMI 和 AMD 公司推出了一种简单编程语言，称为 PALASM，简单地把布尔表达式转化为 PLA 的设置，使得 PAL 和 PLA 的编程相对简单。

早期的可编程器件只允许一次性编程。后来的技术创新，使得可编程逻辑器件发展成可擦写。早先是使用紫外线擦除可编程逻辑。紫外线擦除必须把可编程逻辑器件从电路板中取出后再放入紫外线环境中。所以，该技术在线擦除是不可能的。而且，用紫外线擦除需要较长的时间，大概需要 10 ~ 15 分钟。随后出现了电擦除技术。电擦除技术出现后，现场可编程逻辑阵列被推出。它可以更简单快捷地擦除和可重构编程，而不必把集成芯片从电路中取出。

PAL 和 PLA 之后，很快出现了 CMOS 电可擦除可编程逻辑器件（PLD）。虽然 PLD 这个词可以用于代表任何可编程逻辑器件，但是一般只包括常用的 PALCE22V10 在内的一套器件。PLD 内有由门阵列、多路选择器、触发器或其他标准模块组成的宏模块，一个 PLD 甚至可以有好几个这种宏模块。Lattice Semiconductor 也推出了一种类似的容易编程的器件，称为通用阵列逻辑（GAL）。

现在人们把 PLA, PAL, GAL, PLD 和 PROM 都统称为简单 PLD（SPLD），以区别市面上出现的了一种复杂 PLD（CPLD）。顾名思义，CPLD 比 SPLD 的集成度高，它包含 500 ~ 16 000 个逻辑门。CPLD 就是把几个必要的 PLD 放入同一个芯片中，并通过相同的内部连接电路（交叉开关）把它们连接起来。

20 世纪 80 年代后期，Xillinx 开始使用静态随机存储器（RAM）为可编程器件存储配置信息，同时它们推出了一种大型集成度很高的逻辑器件称为 FPGA。与名字的含义正相反，FPGA 中的基本构模块并不是门阵列，而是包含静态 RAM 和多路选择器的更大更复杂的模块。一些 PLD 提供商和门阵列生产厂家很快地涌入这个市场，生产了各式各样的 FPGA 产品，有的使用了可重构技术，有的使用了一次性熔丝编程技术。在最近 15 年中，FPGA 技术一直得到发展，现在已经能生产出含有 500 万门的 FPGA。

可编程逻辑器件都有基本模块阵列，它可以用来实现任何所要的功能。不同的可编程逻辑器件的差别在于它们提供的基本模块和编程能力。表 3.1 给出了各种可编程逻辑器件的比较。FPGA 比 CPLD 更大更复杂。FPGA 的布线资源比那些简单可编程器件复杂得多。因为存在太多的不同布线路径，所以很难预测信号通过的路径。FPGA 比 CPLD 和 SPLD 要贵，编程也费时。在本章中，我们将介绍以各种可编程逻辑器件，包括 SPLD, CPLD 和 FPGA。

在这个领域很多名称和缩写都曾用于指特定的可编程逻辑器件，但是你会发现它们的含义不明确。比如，PAL 和 PLA，它们都是逻辑阵列，PLA 有由可编程 AND 门和可编程 OR 门，而 PAL 只有由可编程 AND 门，这里没有什么其他理由，只有历史原因。当然，对 PAL 和 PLA 也可以起个其他好的名字。但是，对学生来说还是要记住这些名字所指的一般可编程逻辑器件，因为它们还要跟设计师和设计团队进行交流，习惯可以带来交流的方便。

表 3.1　可编程逻辑器件比较

	SPLD	CPLD	FPGA
密度	低 几百个门	低到中，500 ~ 12 000 个门	中到高，3000 ~ 5 000 000 个门
时延	可预测	可预测	不可预测
成本	低	低到中	中到高
主要提供商	Lattice Semiconductor Cypress AMD	Xilinx Altera	Xilinx Altera Lattice Semiconductor Actel
器件家族例子	**Lattice Semiconductor** GAL16LV8 GAL22V10	**Xilinx** CoolRunner XC9500	**Xilinx** Virtex Spartan
	Cypress PALCE16V8	**Altera** MAX	**Altera** Stratix
	AMD 22V10		**Lattice** Mach ECP
			Actel Accelerator

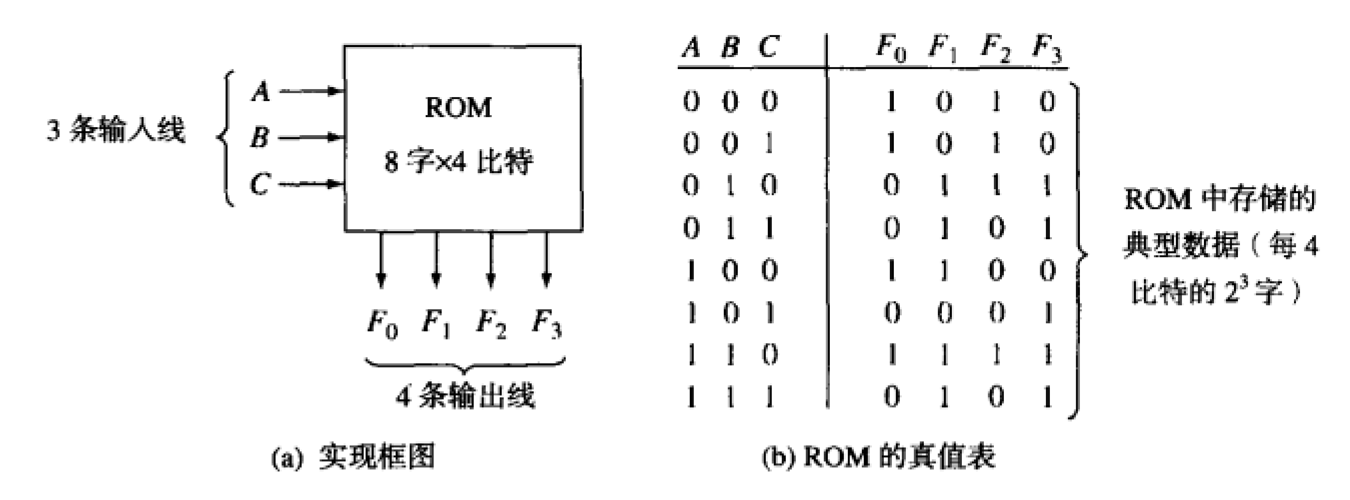
3.2　简单可编程逻辑器件

随着 CPLD 和 FPGA 的发展，早期的可编程逻辑器件，像 ROM, PAL, PLA 和 PLD，都统称为简单可编程逻辑器件（SPLD）。本节中，我们将介绍如用简单 PLD 器件中的电路结构。

3.2.1　只读存储器

只读存储器（ROM）是由一组相互联接的半导体器件阵列组成的，用于存储一组二进制数据。二进制数据一旦存入 ROM 后，便可以随时读出，但一般的操作不能改变已存入的数据。图 3.2(a)是具有 3 位地址输入和 4 位数据输出的 ROM，图 3.2(b)是该 ROM 的真值表，表示输入-输出关系。若输入任意值的 3 位地址，相应的 0 和 1 数据就出现在 4 位输出线上。例如，当输入 $ABC = 010$ 时，数据输出线上的数据为 $F_0F_1F_2F_3 = 0111$。要输出的 ROM 中的每一组数据称为一个字。因为 ROM 有 3 条输入线，所以共有 $2^3 = 8$ 种不同的输入值。每个输入值可以视为一个地址，用于选择存储在 ROM

中的 8 个字之一。因为该 ROM 有 4 条输出线，每个字为 4 位长，所以该 ROM 的大小为 8 字×4 位。

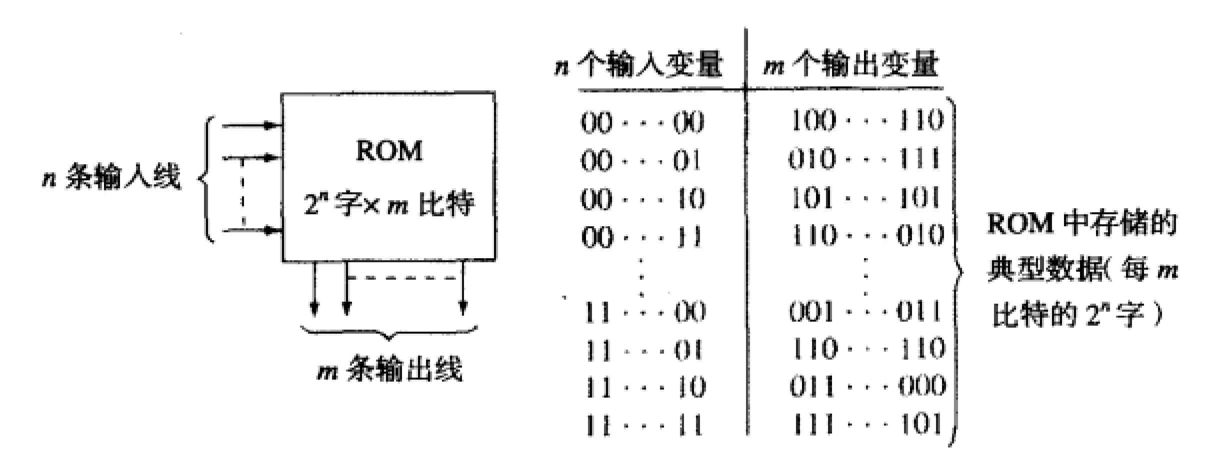


A B C	F_0 F_1 F_2 F_3
0 0 0	1 0 1 0
0 0 1	1 0 1 0
0 1 0	0 1 1 1
0 1 1	0 1 0 1
1 0 0	1 1 0 0
1 0 1	0 0 0 1
1 1 0	1 1 1 1
1 1 1	0 1 0 1

(a) 实现框图　　(b) ROM 的真值表

图 3.2　一个 8 字×4 比特的 ROM

一个 ROM，若有 n 条输入线和 m 条输出线，则它含有 2^n 个字，每个字有 m 位长。输入线作为地址从 2^n 个字中选择一个。当输入的地址代码加到 ROM 时，对应的存储单元中的数据字便出现在输出线上。以图 3.3 为例，若 ROM 的输入（地址线）为 00…11，字 110…010 将被选中，并传送到输出线上。一个 $2^n \times m$ 的 ROM 可以实现有 n 个变量的 m 个函数，因为它可以存储 2^n 行 m 列的真值表。现在市面上可以买到的 ROM，其大小从 32 字×4 位到 512K 字×8 位，或者更大。



n 个输入变量	m 个输出变量
00···00	100···110
00···01	010···111
00···10	101···101
00···11	110···010
⋮	⋮
11···00	001···011
11···01	110···110
11···10	011···000
11···11	111···101

图 3.3　具有 n 个输入和 m 个输出的只读存储器

一个 ROM 主要由一个译码器和一个存储阵列组成。当一组 n 位二进制数加到译码器的输入端时，译码器的 2^n 个输出端仅有一个为 1。这条译码器输出线选出存储阵列中的一个字，这个字的二进制数值就传送到该存储器的输出端。

ROM 的基本类型有掩模可编程 ROM、用户可编程 ROM（PROM）、可擦除可编程的 ROM（统称为 EPROM）、电可擦除可编程 ROM（EEPROM）和 Flash 储存器。掩模可编程 ROM 的数据，在工厂制造时就永久地存储。这是靠有选择地加入或去除存储阵列中行列交界处的开关单元来实现的。这需要准备一个特殊的“掩模”，它在集成电路的制造过程中使用。使用这种掩模是很昂贵的，所以只有在需要大量（一般来说在几千或是更多）的掩模可编程 ROM 时，它才是经济可行的。另外，还有用户一次性可编程 ROM 或 PROM。

在数字系统的开发阶段，常常需要修改 ROM 中存储的数据，所以经常用 EPROM，而不是用掩模可编程 ROM。EPROM 使用一种特殊的电荷存储机制使存储阵列中的开关单元接通或断开。EPROM 编程器用适当的电压脉冲，在存储阵列中存储电荷。用这种方式存储的数据除非使用紫外灯照射，否则将永久保留。擦除后，新的一组数据便可存于 EPROM 中了。

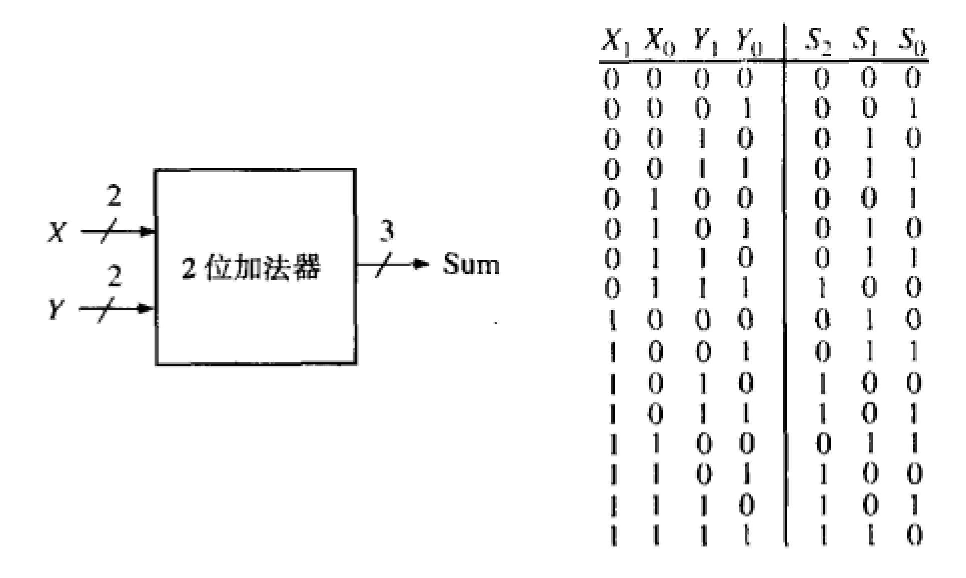
EEPROM 与 EPROM 相似，区别在于其数据的擦除是靠电脉冲而不是紫外线。EEPROM 只能擦除和重新编程有限次，一般是 100 ~ 1000 次。Flash 储存器与 EEPROM 类似，区别在于它们

使用一种不同的电荷存储机制。它们通常有内嵌的编程和可擦除能力，这样数据便可在电路中被写入，而不需要单独的编程器件。

一个 ROM 可以用来实现任何组合电路。首先，把所有输入组合对应的输出都存储在 ROM 中，然后对应任意输入，我们可以通过查找 ROM 中存储的输出表来得到输出的值。所以，基于 ROM 的实现也称为**查表法**（LUT）。

下面我们用 ROM 实现一个二位加法器，完成两个 2 位二进制数的加法。因为 2 位二进制数的最大值为 3，所以和的最大值为 6，需要用 3 位二进制数表示。该加法器的真值表参见图 3.4。我们也设计一个二位全加器，除了两个 2 位二进制数的输入外，还需要一个进位数。

2 位二进制数的加法器可以使用一个 16×3 的 ROM 来实现。输入（X 和 Y）必须与 4 条地址线相连，三条输出线将产生和数。



X_1	X_0	Y_1	Y_0	S_2	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

图 3.4　一个 2 位加法器模块及其真值表

图 3.5 给出了用 ROM 实现二位全加器的例子。假设连接如图 3.5 所示，则 ROM 中 16 个存储单元的值分别为 0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5 和 6（十进制表示）。和数的最低位来自数据总线的最低位。

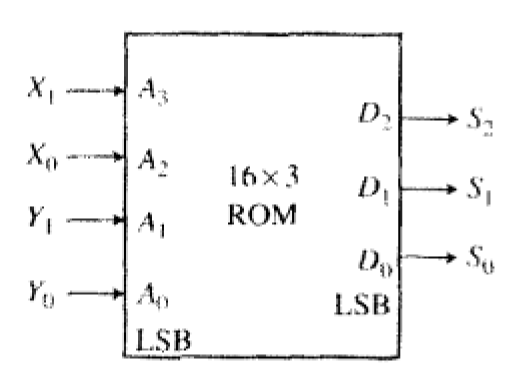
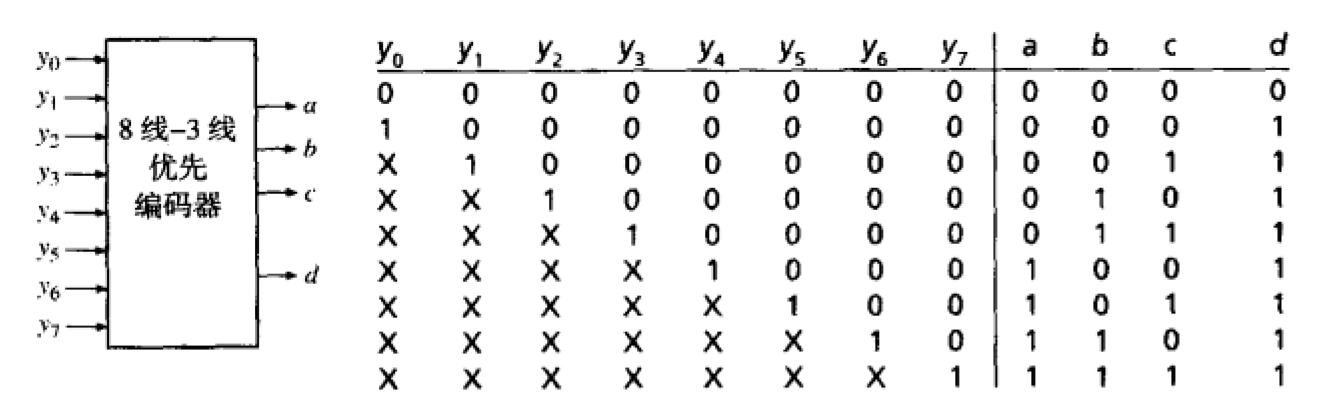


图 3.5　一个 2 位全加器的 ROM 实现

例 1　若用 ROM 实现一个 8 线–3 线优先编码器，则需要多大的 ROM?

解： 编码器是译码器的反过程，图 3.6 给出 8 线–3 线优先编码器。若输入 y_i 为 1，则其他输入均为 0，输出 abc 所表示二进制数等于 i。另一个输出 d 表示输出的有效性，若 $d = 1$ 则表示输出比特 a, b, c 有效。如果输入中有多个 1，则最高位的 1 用于决定输出值。输入输出的真值表参见图 3.6。真值表中的 X 表示随意项。由此可以看出，8 线–3 线优先编码器有 8 个输入和 4 个输出，因此实现它需要一个 $2^8×4$ 比特的 ROM。

说明： 该此 ROM 中共有 256 个存储单元。如果把图 3.6 的真值表中的所有随意项都考虑在内，那么确实需要 256 个单元。



y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	a	b	c	d
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
X	1	0	0	0	0	0	0	0	0	1	1
X	X	1	0	0	0	0	0	0	1	0	1
X	X	X	1	0	0	0	0	0	1	1	1
X	X	X	X	1	0	0	0	1	0	0	1
X	X	X	X	X	1	0	0	1	0	1	1
X	X	X	X	X	X	1	0	1	1	0	1
X	X	X	X	X	X	X	1	1	1	1	1

图 3.6 8 线–3 线优先编码器

例 2 用 ROM 实现一个时序机，其状态表参见图 3.7。你已经看到该时序机就是第 1 章中设计的 BCD–余 3 码转换器。

	当前状态		Z 下一状态	
	$X=0$	$X=1$	$X=0$	$X=1$
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

图 3.7 一个连续电路的状态表

解：使用 ROM 和触发器可以容易地设计时序电路。时序电路的组合逻辑部分可以用 ROM 实现。ROM 可用来实现输出函数和下一状态函数。电路状态可以存储在 D 触发器中，然后反馈给 ROM 的输入。使用 D 触发器比 J-K 触发器好，因为 J-K 触发器需要更多的来自 ROM 的输入。虽然 D 触发器的输入方程比 J-K 触发器需要更多的逻辑门，但这无关紧要，因为 ROM 的大小仅取决于输入和输出比特数，而不取决于要实现的方程的复杂度。因此，状态赋值方法也不重要，直接二进制状态赋值跟其他方法一样好。

为了实现上述时序电路，需要用一个 ROM 和三个 D 触发器。根据当前状态和输入 X，ROM 将生成下一状态和输出 Z。因此，ROM 需要 4 条地址线（3 条来自触发器，1 条来自输入 X）和 4 条输出线（3 条表示下一状态，1 条表示输出 Z）。图 3.8 给出了电路的具体实现。因为 ROM 有 4 个比特输入，所以其含有 $2^4=16$ 个字。通常，有 i 个比特输入、j 个比特输出和 k 个状态变量的 Mealy 时序电路，可以用 k 个 D 触发器和一个有 $i+k$ 个输入（2^{i+k} 个字）和 $j+k$ 个输出的 ROM 来实现。

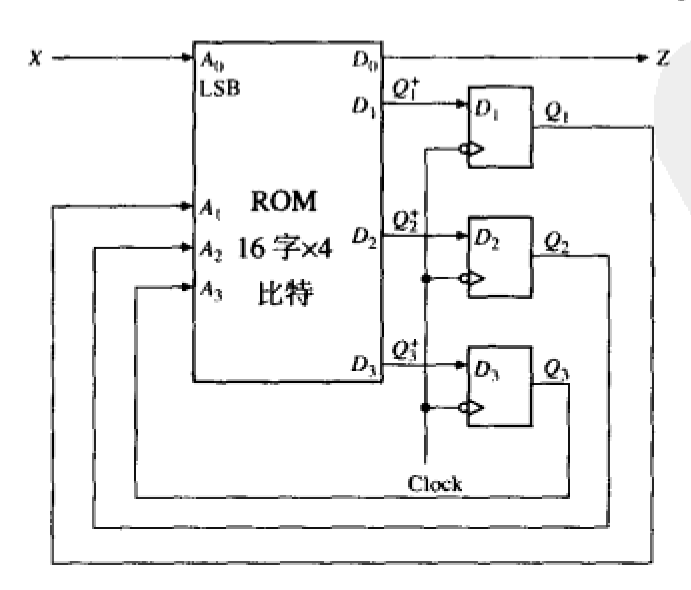


图 3.8 使用 ROM 实现一个 Mealy 连续电路

下面我们推导 ROM 中存储的数据内容。表 3.2 给出了该时序电路的真值表，它将实现图 3.7 的状态表，用 0 代替了状态表中的所有随意项，采用了直接二进制状态赋值。

表 3.2　ROM 真值表

Q_3	Q_2	Q_1	X	Q_3^+	Q_2^+	Q_1^+	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	1	1
0	0	1	1	1	0	0	0
0	1	0	0	1	0	0	0
0	1	0	1	1	0	0	1
0	1	1	0	1	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1
1	1	0	0	0	0	0	1
1	1	0	1	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0

假设 Q_3, Q_2, Q_1 和 X 按照如此顺序接入地址线，则 X 与最低有效位连接。实现该时序机的 ROM 中的数据为 3, 4, 7, 8, 8, 9, A, B, B, C, 0, 1, 1, 0, 0, 0（十六进制表示）。十六进制是一种表示输出的简明方便的方法。输出 Z 可以从数据线的最低有效位得到。而下一状态信息可以从 ROM 数据线的三个最高有效位得到。

3.2.2　可编程逻辑阵列

可编程逻辑阵列（PLA）的基本功能与 ROM 是相同的。有 n 个输入和 m 个输出的 PLA（参见图 3.9）可以实现 m 个 n 变量函数。PLA 的内部结构与 ROM 不同，与门（AND）阵列代替了译码器，用于选择输入变量的乘积项，或门（OR）阵列再把这些乘积项或起来实现输出逻辑函数。

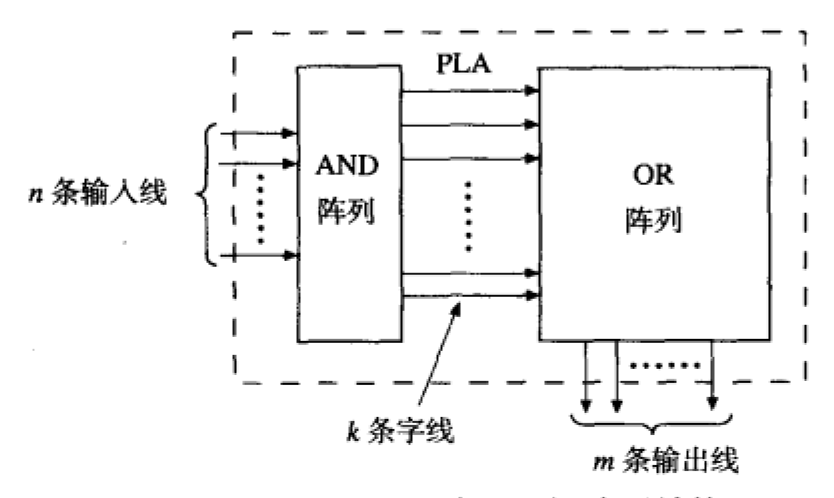


图 3.9　可编程逻辑阵列结构

图 3.10 给出了一个 PLA，实现下列逻辑函数：

$$F_0 = \sum m(0,1,4,6) = A'B' + AC' \tag{3.1}$$
$$F_1 = \sum m(2,3,4,6,7) = B + AC'$$
$$F_2 = \sum m(0,1,2,6) = A'B' + BC'$$
$$F_3 = \sum m(2,3,5,6,7) = AC + B$$

上面的逻辑函数有三个变量。在一个 PLA 实现中，先产生表达式中的乘积项，随后把这些乘积项用用或门或起来。所以在 PLA 实现中，这些乘积项是可以共用的。我们不用单独化简每个函数表达式，而是对整个乘积项的数目进行化简。上面 4 个表达式中有 5 个不同的乘积项。图 3.10 示出了实现该逻辑函数的含有 5 个乘积项的 3 输入 4 输出 PLA 结构。注意到每个表达式中乘积项的数目并不重要，只要我们把输出所需的所有乘积项用与门生成就可以。

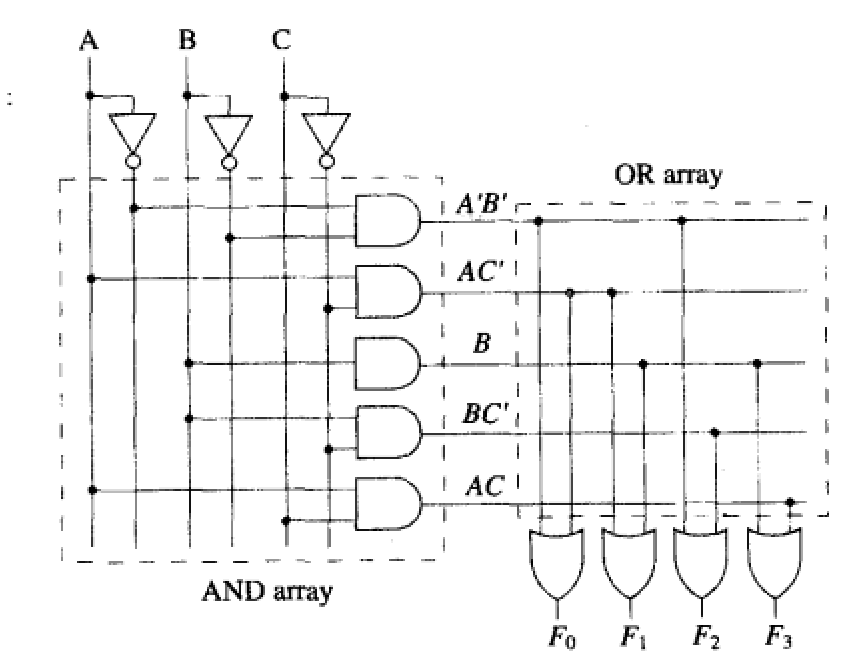


图 3.10　具有 3 输入、5 个乘积项和 4 个输出的 PLA 结构（逻辑层面）

在 PLA 内部，它可能使用的是或非–或非逻辑，而不是与或逻辑。图 3.10 表示的阵列与图 3.11 给出的 nMOS PLA 结构是等效。阵列中的逻辑门是通过行和列间的 nMOS 开关晶体管形成。

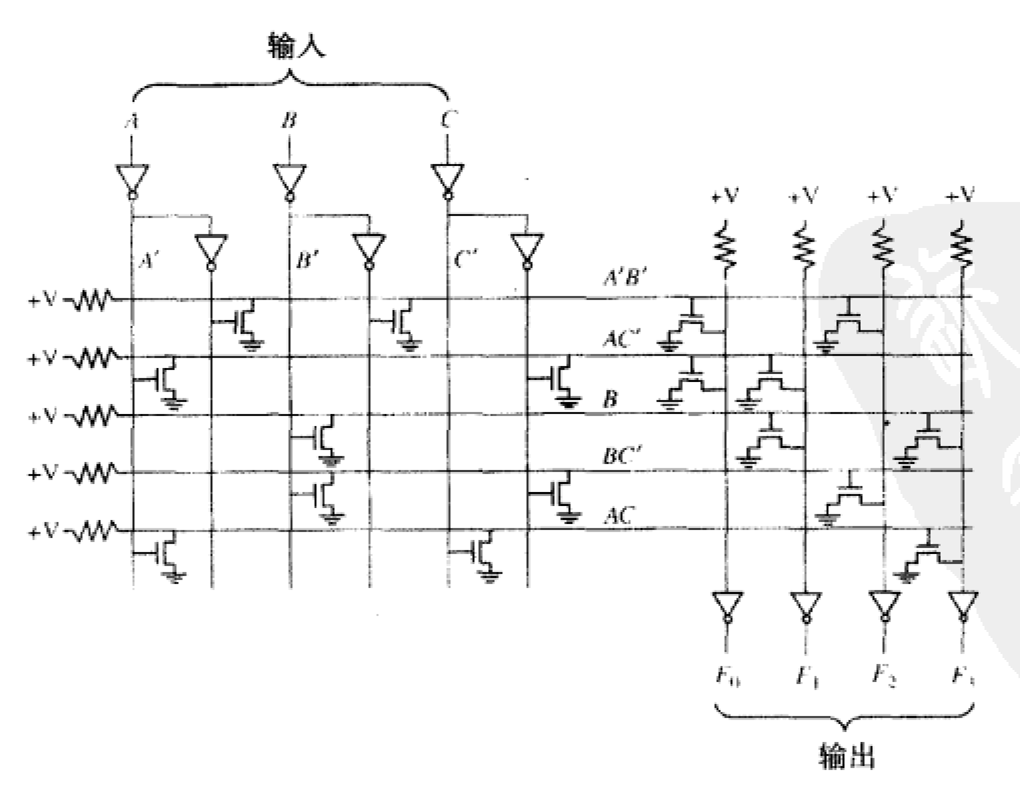


图 3.11　具有 3 输入、5 个乘积项和 4 个输出的 PLA 结构（晶体管层面）

金属氧化物半导体晶体管（MOS）的三个极分别为源极、漏极和栅极。栅极通常用来控制晶体管的通和断。MOS 晶体管有两种：nMOS 和 pMOS。本节中我们使用了 nMOS 晶体管。20 世纪 90 年代起流行互补 MOS（CMOS）技术，nMOS 和 pMOS 晶体管都用于互补方式。

图 3.12 给出了用 nMOS 管实现的一个两输入或非门。晶体管起开关作用，若该逻辑门输入为逻辑 0，则晶体管关闭；若门的输入为逻辑 1，则晶体管提供一条到地的导通路径。如果 $X_1 = X_2 = 0$，两个晶体管都关闭，上拉电阻把 Z 输出置成逻辑 1 高电位（+V）。如果 X_1 和 X_2 其中有一个为高电平 1，则相应的晶体管导通，$Z = 0$。这样，$Z = (X_1 + X_2)' = X_1' X_2'$，对应一个或非门。实现 F_0 的 PLA 阵列等同于如图 3.13 所示的或非-或非门结构。去除掉额外的反相器后，就简化为一个与或结构了。

图 3.12　nMOS NOR 门

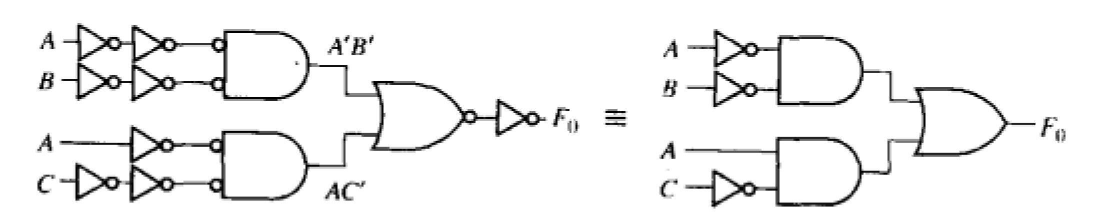


图 3.13　从 NOR-NOR 到 AND-OR 的转换

一个 PLA 的内容可以用一个修改的真值表来表示。表 3.3 给出了图 3.10 的 PLA。表格的输入部分代表乘积项。符号 0, 1 和 — 分别表示在乘积项中一个变量是否取补或者不存在。表格的输出部分给出了在每个输出函数中出现的乘积项，用 1 或 0 表示该乘积项是否在相应的输出函数中出现。这样，表 3.3 中的第一行表示乘积项 $A'B'$ 在输出函数 F_0 和 F_2 中出现，第二行代表乘积项 AC' 在 F_0 和 F_1 中出现。

表 3.3　式(3.1)的 PLA 表

乘积项	输入			输出			
	A	B	C	F_0	F_1	F_2	F_3
$A'B'$	0	0	—	1	0	1	0
AC'	1	—	0	1	1	0	0
B	—	1	—	0	1	0	1
BC'	—	1	0	0	0	1	1
AC	1	—	1	0	0	0	1

下面我们将用 PLA 实现下面的函数：

$$F_1 = \sum m(2,3,5,7,8,9,10,11,13,15) \tag{3.2}$$
$$F_2 = \sum m(2,3,5,6,7,10,11,14,15)$$
$$F_3 = \sum m(6,7,8,9,13,14,15)$$

如果我们单独把各个函数化简，结果是

$$F_1 = bd+b'c+ab' \tag{3.3}$$
$$F_2 = c+a'bd$$
$$F_3 = bc+ab'c'+abd$$

如果我们用 PLA 实现这些简化了的方程，则一共需要 8 个不同的乘积项（包括 C）。

下面我们要缩减 PLA 表中的行数，而不是单独地化简每一个函数。在这种情况下，因为 PLA 的大小不取决于乘积项的数量，所以每个表达中乘积项的数量就不那么重要了。图 3.14 示出了用卡诺图描述的表达式(3.3)。因为项 $ab'c'$已经出现在 F_3 中，所以在 F_1 用它取代 ab'。因为 ab'所覆盖的的另外两个 1 被 $b'c$ 所覆盖，这就意味着与 ab'对应的 PLA 表格中的一行没有必要，可以删去。因为乘积项 $a'bd$ 和 abd 分别在 F_2 和 F_3 中要用，所以我们可以用 $a'bd + abd$ 取代 F_1 中的 bd。这样也就不必用 PLA 表格的一行来表示 bd 了。因为 $b'c$ 和 bc 分别在 F_1 和 F_3 中需要用，所以我们可以用 $b'c + bc$ 代替 F_3 中的 c。最后的表达式(3.4)对应于简化的 PLA 表格（参见表 3.4）。可以使用 Espresso 算法代替卡诺图来减少行数。这个复杂算法在 Brayton[12]所著的 *Logic Minimization Algorithms for VLSI Synthesis* 中有介绍。

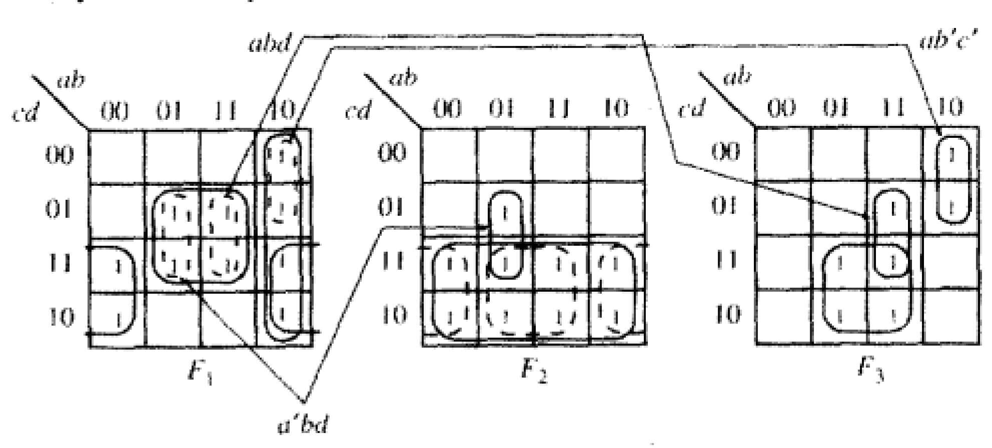


图 3.14 多输出卡诺图

表 3.4 化简后的 PLA 表

A	b	c	d	F1	F2	F3
0	1	—	1	1	1	0
1	1	—	1	1	0	1
1	0	0	—	1	0	1
—	0	1	—	1	1	0
—	1	1	—	0	1	1

$$F_1 = a'bd+abd+ab'c'+b'c \tag{3.4}$$
$$F_2 = a'bd+b'c+bc$$
$$F_3 = abd+ab'c'+bc$$

式(3.4)只有 5 个不同的乘积项，所以 PLA 表只有 5 行。这是对式(3.3)的一个很有意义的改进，它需要 8 个乘积项。图 3.15 给出有 4 个输入、5 个乘积项和 3 个输出的该 PLA 结构。图 3.15 中字线和输入或输出线相交处的一个黑点，代表阵列中的一个开关单元。

一个 PLA 表与 ROM 的真值表很不一样。在真值表中，每一行代表了一个最小项，每一个输入组合恰好能选中一行，选中行的输出部分的 0 和 1 决定相应的输出值。另一方面，PLA 表中的每一行代表了一个乘积项，所以零行、一行或更多的行可以被每一个输入组合选中。要确定给定输入组合的 F 值，PLA 表中被选定行的 F 值必须或（OR）在一起。在下面的例子中，参考表 3.4 给出的 PLA 表。如果 $abcd = 0001$，则没有一行被选中；所有的 F_i 的值是 0。如果 $abcd = 1001$，

则只有第三行被选中，$F_1F_2F_3 = 101$。如果 $abcd = 0111$，则第一行和第五行被选中。因此，$F_1 = 1 + 0 = 1$，$F_2 = 1 + 1 = 1, F_3 = 0 + 1 = 1$。

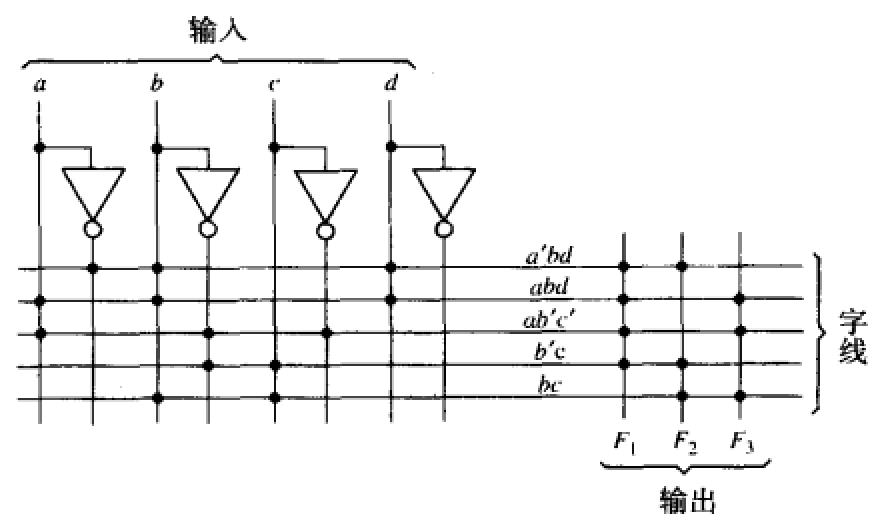


图 3.15　式(3.4)的 PLA 实现

下面我们用 PLA 和 3 个 D 触发器来实现如图 1.23 所示的 BCD–余 3 码转换器时序电路。电路结构与图 3.8 相同，区别仅在于 ROM 被 PLA 取代。基于如图 1.25 所示的表达式，我们得到所需的 PLA 表（参见表 3.5）。

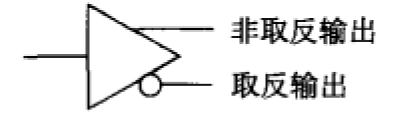
表 3.5　PLA 表

乘积项	Q_1	Q_2	Q_3	X	Q_1^+	Q_2^+	Q_3^+	Z
Q_2'	—	0	—	—	1	0	0	0
Q_1	1	—	—	—	0	1	0	0
$Q_1Q_2Q_3$	1	1	1	—	0	0	1	0
$Q_1Q_3'X'$	1	—	0	0	0	0	1	0
$Q_1'Q_2'X$	0	0	—	1	0	0	1	0
$Q_3'X'$	—	—	0	0	0	0	0	1
Q_3X	—	—	1	1	0	0	0	1

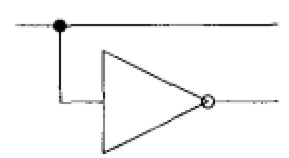
3.2.3　可编程阵列逻辑

可编程阵列逻辑（PAL）是可编程逻辑阵列（PAL）的特例，因为它的 AND 阵列是可编程的，OR 阵列是固定的。PAL 的基本结构与如图 3.9 所示的 PLA 相同。因为只有 AND 阵列是可编程的，所以 PAL 比 PLA 便宜，并且 PAL 更容易编程。由于这个原因，逻辑函数经常使用 PAL 来取代单独的逻辑门，用于实现多个逻辑函数。

图 3.16(a)表示一个还没有编程的 PAL 片段（Segment）。符号



代表输入缓冲，逻辑上等效于



因为每个 PAL 输入必须驱动许多 AND 门输入，所以要使用缓冲器。PAL 编程是对一些相互连接的节点进行编程，使其连接到与门的输入输入。在 PAL 中与门输入的连接点用 × 标出，如下图所示：

作为一个例子，我们用图 3.16(a)的 PAL 片段实现函数 $I_1I_2' + I_1'I_2$。图 3.16(b)中的 × 代表 I_1 和 I_2' 已连接到第一个 AND 门，I_1' 和 I_2 连接到另一个 AND 门。

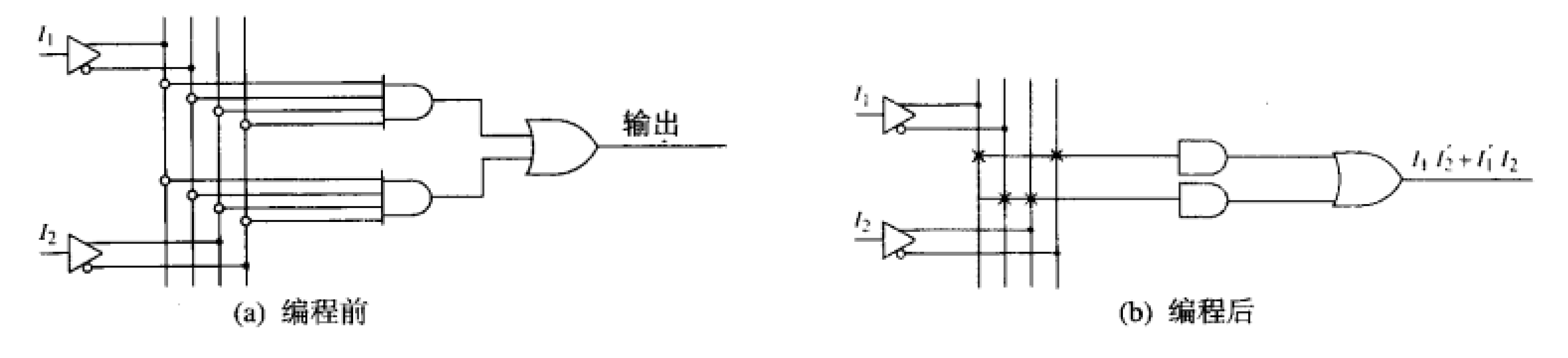


图 3.16　PAL 一个片段（Segment）

当基于 PAL 进行设计时，我们必须把逻辑表达式化简，使之能用一个或多个 PAL 来实现。与更一般的 PLA 不同，PAL 中的一个 AND 门不能被两个或两个以上的 OR 门共用。所以，每个逻辑表达式必须单独进行化简，不用考虑彼此间的公共项。对于一种给定的 PAL，与或门（OR）相连的 AND 门的个数是固定的和有限的。如果化简后的表达式中含有较多的与运算，则我们不得不选择具有更多输入端和较少输出端的 PAL。

下面我们以全加器为例，介绍如何进行 PAL 编程。全加器的逻辑表达式为

$$Sum = X'Y'C_{in} + X'YC'_{in} + XY'C'_{in} + XYC_{in}$$

$$C_{out} = XC_{in} + YC_{in} + XY$$

如图 3.17 给出了 PAL 的一部分，其中每个或门都由 4 个与门驱动，对 PAL 编程产生图中×表示的输入连接，用以实现全加器的逻辑表达式。例如，第一行的×表示乘积项 $X'Y'C_{in}$ 的实现。

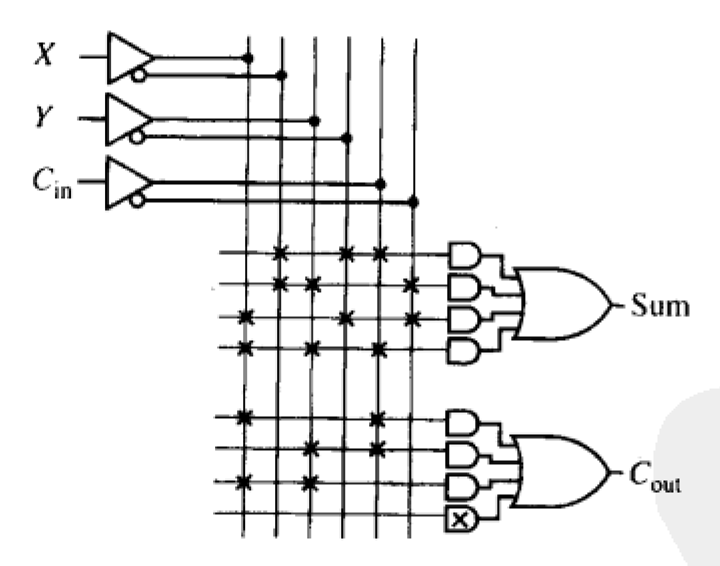


图 3.17　用 PAL 实现一个全加器

典型的组合 PAL 有 10 ~ 20 个输入、2 ~ 10 个输出，并且每一个 OR 门由 2 ~ 8 个 AND 门驱动。PAL 中也可有 D 触发器，其输入由可编程阵列逻辑驱动。这种 PAL 称为时序 PAL，可以用来方便地实现时序电路。图 3.18 给出了时序 PAL 的一个片段。D 触发器由 OR 门驱动，该 OR 门又由两个 AND 门驱动。触发器的输出通过一个缓冲器被反馈到可编程 AND 阵列。这样，AND 门的输入可以连到 A, A', B, B', Q 或 Q'。由此我们可以得出，图 3.18 中所示结构实现了下一状态表达式：

$$Q^{+} = D = A'BQ' + AB'Q$$

触发器的输出与一个反相三态缓冲器连接，当 EN = 1 时此三态缓冲器有效。

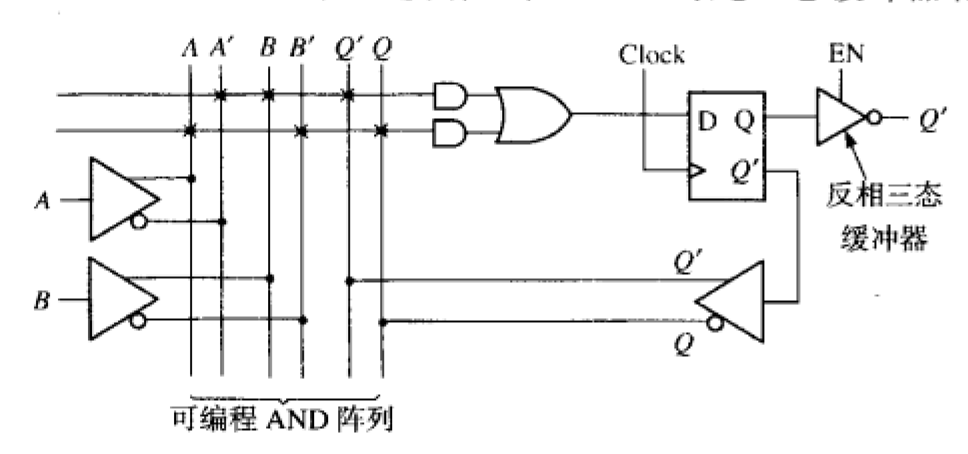


图 3.18 一个时序 PAL 的一个片段

在几十年以前，数字系统设计者很喜欢使用 PAL 进行设计。当时应用最广泛的 PAL 是 16R4，它有 16 个输入变量的 AND 门阵列和 4 个 D 触发器。现在，我们有很多其他种类的可编程器件可以使用，如 GAL, CPLD 和 FPGA 等。PAL 现在已经很少见了，所以我们对传统 PAL 器件就不再多加描述了。

3.2.4 可编程逻辑器件/通用阵列逻辑

PAL 和 PLA 一般用于实现所需要的简单电路和接口逻辑。随着集成电路技术的发展，现在涌现出很多其他的可编程逻辑器件可用。传统的 PAL 不能可重构编程，但是现在出现了 Flash 电可擦除可重构编程的 PAL，一般称为可编程逻辑器件（PLD）。

22CEV10（参见图 3.19）是一款 CMOS 电可擦除 PLD，它可以用于实现组合电路和时序电路。PLD 这一缩写广义上指所有可编程逻辑器件，狭义上指像 22CEV10 这样的器件。大多数 PLD 除了含有 PAL 中的与-或阵列之外，还有某种宏模块，该模块含有多路选择器并具有一些其他的可编程性。这些 PLD 是按照它们的输入和输出能力来命名的，比如 22CEV10 就有 12 个固定输入管脚和 10 个可以为输入或者输出编程的管脚，它有 10 个 D 触发器和 10 个或门。每个或门可以连接 8 到 16 个与门来驱动，而每一个或门驱动一个输出逻辑宏单元（Output Logic Macrocell）。一共有 10 个宏单元，每一个宏单元含有一个 D 触发器。这些 D 触发器具有同一个时钟、同一个异步复位（AR）输入和同一个同步预置（SP）输入。22V10 这个名字表示有 22 个输入和输出管脚的多用途 PAL，其中 10 个管脚是双向 I/O（输入/输出）管脚。

图 3.20 表示 22CEV10 的输出宏单元内部结构。我们通过对这些宏单元的编程来控制与输出管脚的相连。输出多路选择器(MUX)的控制输入 S_1 和 S_0 用于选择 MUX 的一个输入数据。例如，若 S_1S_0=10，则选中数据输入 2。每个宏单元有两个可编程连接节点位，当相应的节点位被编程时，S_1 或 S_0 就接地（逻辑 0）。如果要擦除该节点位，则使接地的控制线（S_1 或 S_0）断开，并使它上浮到逻辑 1。当 S_1= 1 时，或门输出不经过触发器，通过多路选择器和输出缓冲器连到 I/O 管脚。或门输出也可以反馈到与门的输入端。如果 S_1= 0，则触发器输出与输出管脚相连，它也可以反馈回到与门的输入端。当 S_0= 1 时，输出不反相，输出高电平有效。当 S_0 = 0 时，输出反相，输出低电平有效。输出管脚由三态反相缓冲器驱动。当缓冲器输出处在高阻状态时，或门和触发器与输出管脚断开，这时该管脚可以用做输入端口。图 3.20(a)上的虚线给出了当 S_1 和 S_0 都是 0 时通过输出宏单元的路径，图 3.20(b)中的虚线给出了当 S_1 和 S_0 都是 1 时通过输出宏单元的路径。注意到，在第一种情况下，触发器输出 Q 被输出缓冲器取反；在第二种情况下，或门输出被取反两次，所以相当于未取反。

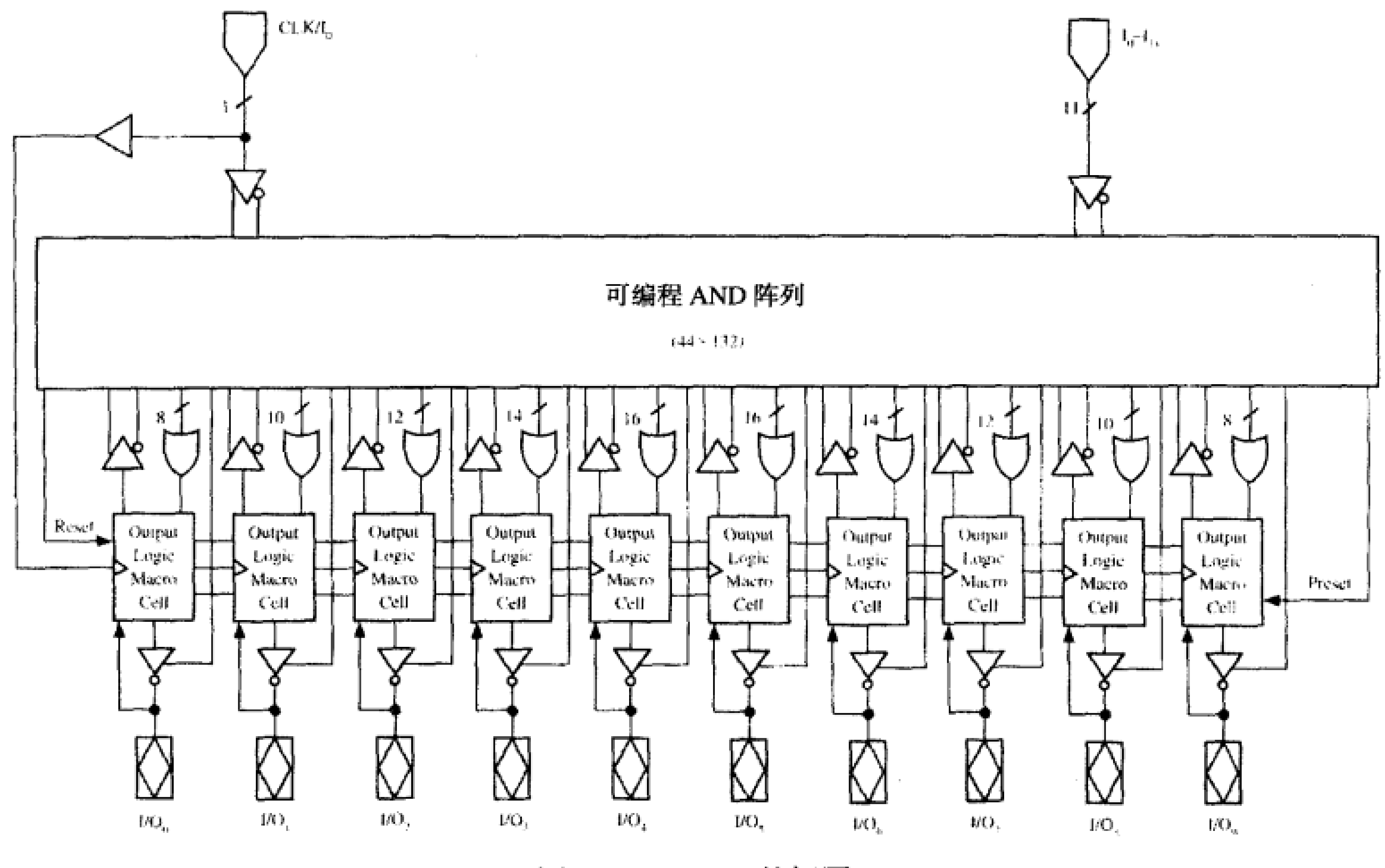
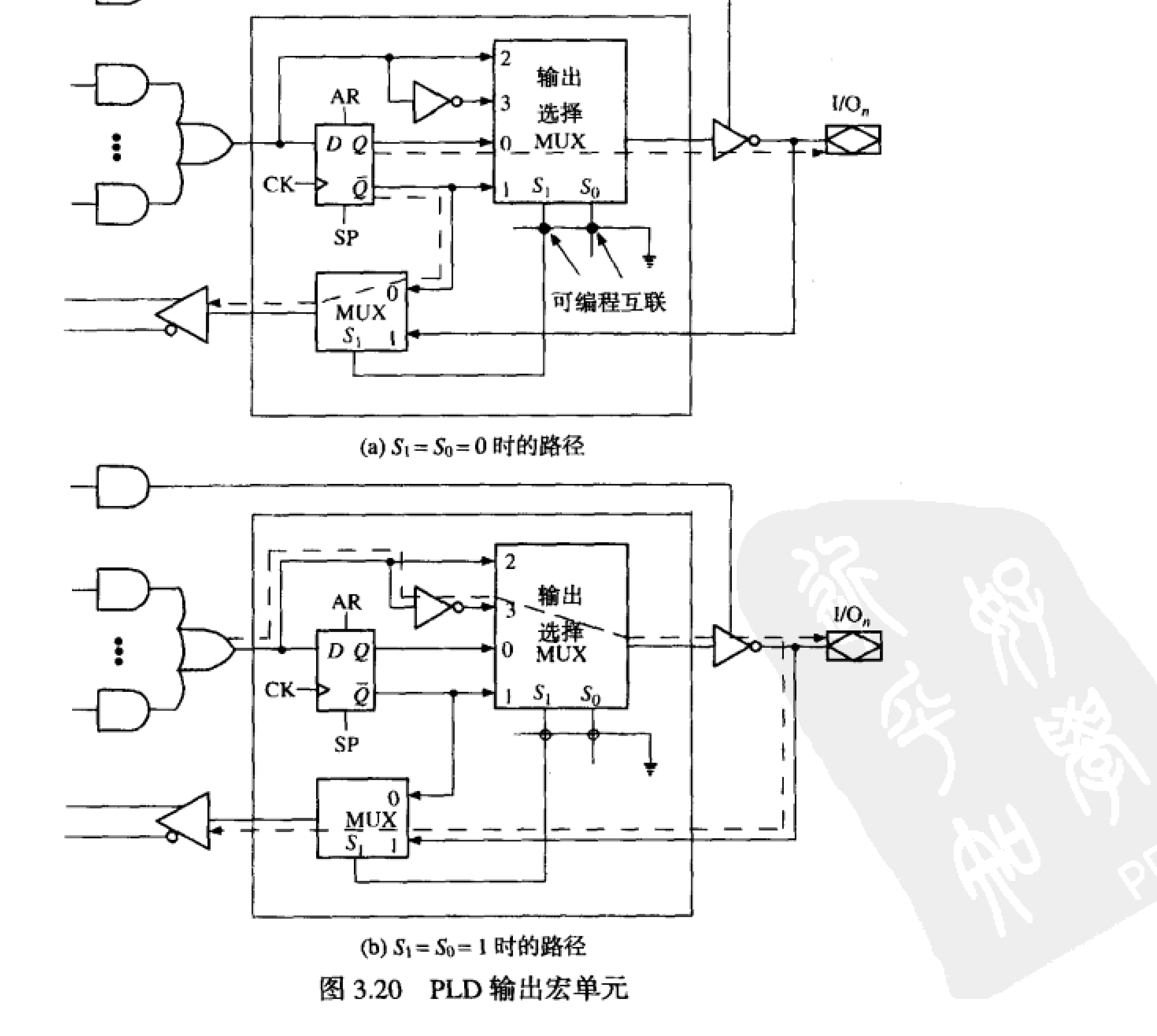


图 3.19　22V10 的框图



(a) $S_1 = S_0 = 0$ 时的路径

(b) $S_1 = S_0 = 1$ 时的路径

图 3.20　PLD 输出宏单元

与 22V10 特性类似的一些 PLD 比较普及，一般都有 8 ~ 12 个 I/O 管脚，每个输出管脚都与一个输出宏单元相连，而每个宏单元都有一个 D 触发器。这些 I/O 管脚都可被编程为输入端口，或者为与组合或触发器相连的输出端口。一些 PLD 有专门的时钟输入端口，而其他的 PLD 有既可以做时钟输入又可以做一般输入的两用管脚。所有 PLD 输出端都有三态缓冲器，其中一些 PLD 还有专用输出使能端（$\overline{OE}$）。

Lattice Semiconductor 也推出了一种类似在线可编程逻辑器件称为通用阵列逻辑（GAL）。GAL 若用于实现一种称为“胶水”逻辑的接口逻辑，则非常理想。像 PALCE22V10 和 PALCE20V8 等大多数 PLD 器件，GAL 器件也有类似的命名，比如 GAL22V10 和 GAL20V8 等。

PLD 的设计流程

现在广泛使用计算机辅助设计工具软件进行 PAL 和 PLD 的编程。这些工具软件的输入可以是逻辑表达式、真值表、状态图或状态表，该软件能够自动地把它们转化为所需的编程二进制数据。然后把这些编程数据下载到 PLD 编程器中，生成所需的连接并验证 PAL 的操作。现有的很多新型的 PLD 可以像 EPROM 和 EEPROM 一样可擦除和可重构编程。因此在这些新型器件中，与所需的 EEPROM 数据相对应的编程数据也是由软件生成的。

PALASM 和 **ABEL** 是两种在 PAL 和 PLD 中常用的编程设计语言。PALASM 是 MMI 和 AMD 公司推出的 PLD 设计语言，ABEL 是 DATA I/O 推出的 PLD 设计语言。Intel 也制造过 PLD 器件，它用了称为 PLDShell 的 PLD 语言。虽然 PALASM 和 ABEL 语言仍可以使用，但是我们用硬件描述语言（如 VHDL 和 Verilog）设计现在的 GA 器件 L。

3.3　复杂可编程逻辑器件

随着集成电路技术的发展，在同一个可编程芯片上可以集成实现几个 PLD 的功能，这种芯片称为复杂可编程逻辑器件（CPLD）。当同一芯片上同时包含存储单元（如触发器等）时，就可以在一片 CPLD 上实现一个小型的数字系统。

从概念上来说，CPLD 是 PAL 的扩展。一般 CPLD 就是把大量几个类似 PAL 的逻辑块集成到一个芯片上，并用一个可编程连接矩阵把它们都连接起来。CPLD 通常包含 500 ~ 10 000 个逻辑门，其内部的各个 PLD 通过类似纵横交叉开关的结构连接在一起。对于一个 $N \times M$ 的纵横交叉开关，它可以使 N 个输入中的每一个同时与任意 M 个输出相连。构造这种开关很昂贵，但是它能使时序变得可以预测。很多 CPLD 都是电可擦除和可重构编程的，所以有时候也称为 EPLD（可擦除 PLD）。

典型的 CPLD 包含很多宏单元，这些宏单元分属不同的功能块，不同的功能块之间通过连接阵列来相互连接。每个宏单元都有一个触发器和一个 OR 门，它们的输入都连接到与门阵列。有些 CPLD 是基于 PAL 的，所以每个或门都有固定的与门与之相连。其他的 CPLD 是基于 PLA 的，在一个功能块中，任意与门输出都可以与该功能块中的任意或门输入相连。Xilinx, Altera, Lattice Semiconductor, Cypress 和 Atmel 是当今主要的 CPLD 生产厂商。市面上的主要产品在表 3.6 中列出。有些厂商根据可用门数给出容量大小，有的则按照逻辑单元数来给出。

表 3.6　主要 CPLD 及其容量

生产厂商	CPLD 家族	包含门的个数
Xilinx	CoolRunner-II	750 ~ 12K
	CoolRunner XPLA3	750 ~ 12K

（续表）

生产厂商	CPLD 家族	包含门的个数
	XC9500XV	800 ~ 6400
	XC9500	800 ~ 6400
	XC9500XL	800 ~ 6400
Atmel	CPLD ATF15	750 ~ 3000 可用门
	CPLD-2 22V10	500 可用门
Cypress	Delta39K	30K ~ 200K
	Flash370i	800 ~ 3200
	Quantum38K	30K ~ 100K
	Ultra37000	960 ~ 7700
	MAX340 high-density EPLD	600 ~ 3750
Lattice	ispXPLD 5000MX	75K ~ 300K
Semiconductor	ispMACH 4008B/C/V/Z	640 ~ 10 240
Altra	MAX II	240 ~ 2210 个逻辑单元
	MAX3000	600 ~ 10K 可用门
	MAX7000	600 ~ 10K 可用门

3.3.1 CPLD 示例：Xilinx 公司的 CoolRunner 系列芯片

Xilinx 主要生产两个系列的 CPLD 芯片：CoolRunner 和 XC9500。图 3.21 为 Xilinx XCR3064XL 芯片（隶属 CoolRunner 系列 CPLD）的基本结构。这种 CPLD 有 4 个功能块，每个功能块包含有 16 个宏单元（MC1，MC2，…）。每个功能块都是一个可编程与或门阵列，其配置与 PLA 相同。每个宏单元都含有一个触发器和几个多路选择器，多路选择器用于将信号从功能块送到输入/输出（I/O）块或**互联阵列**（Interconnect Array, **IA**）。**互联阵列**从宏单元输出或 I/O 块中选择信号，然后将信号反馈到功能块的输入端。这样一个功能块生成的信号可以作为其他功能块的输入信号。I/O 块为 IC 上的双向 I/O 引脚与 CPLD 内部的相连提供了接口。

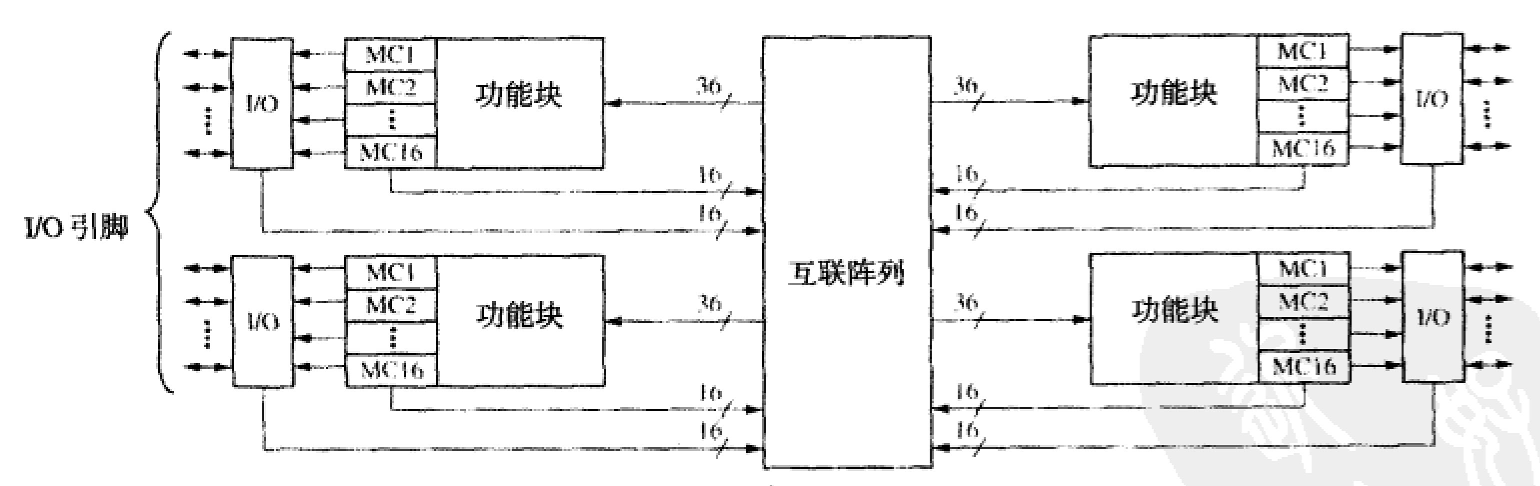


图 3.21 Xilinx CoolRunner XCR3064XL CPLD 的结构

图 3.22 说明了 PLA（功能块）产生的信号如何通过宏单元传到 I/O 引脚。IA 的 36 个输出端（或其反相端）均可以与 48 个 AND 门中的任意输入端相连。每一个或门最多可以连接 48 个由与阵列产生乘积项。图中的宏单元逻辑是实际电路的一个简化表示。第一个 Mux(1)可通过编程选择或门的输出（或反相输出）。宏单元输出端的 Mux(2)可以通过编程选择组合电路输出端（G）或者触发器的输出端（Q）。这个输出连接了互联阵列和输出单元。输出单元包含一个用于驱动 I/O 引脚的三态缓冲器(3)。缓冲器的使能输入端可以通过编程由多个信号源来控制。当 I/O 引脚用作输入时，该缓冲器必须

置为无效。

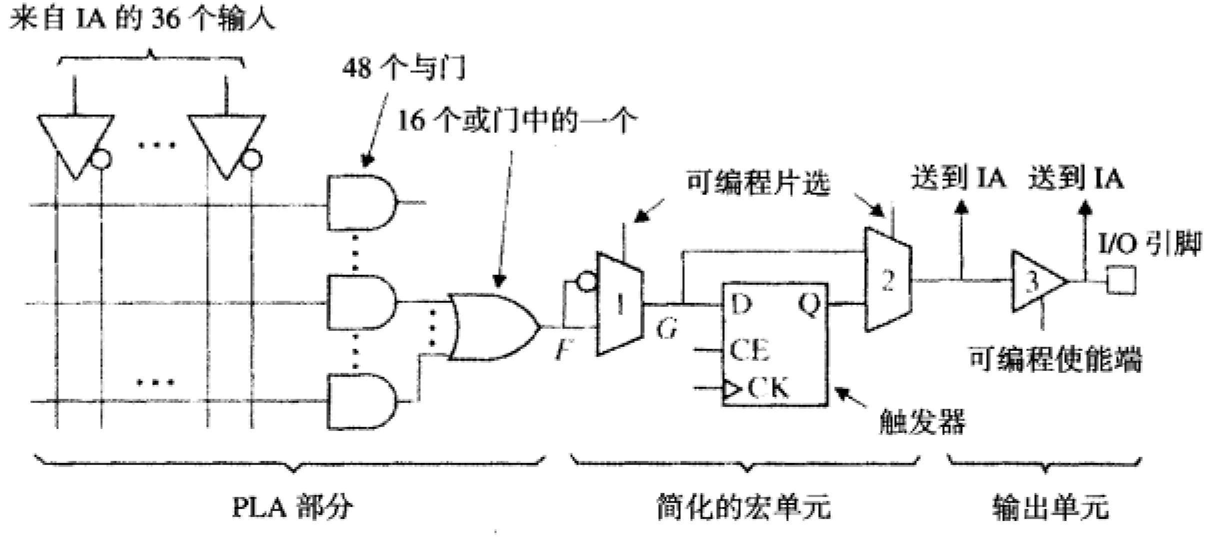


图 3.22　CPLD 功能块和宏单元（XCR3064XL 的简化图）

图 3.23 说明了如何用一个 CPLD 实现一个由两个输入、两个输出和两个触发器构成的 Mealy 时序电路。实现该电路需要四个宏单元，两个宏单元用来生成触发器的两个 D 输入，另外两个宏单元则用来生成两个 Z 输出。触发器的输出通过互联矩阵（没有画出）反馈给与门阵列的输入端。所需的乘积项数取决于 D 和 Z 逻辑表达式的复杂度。

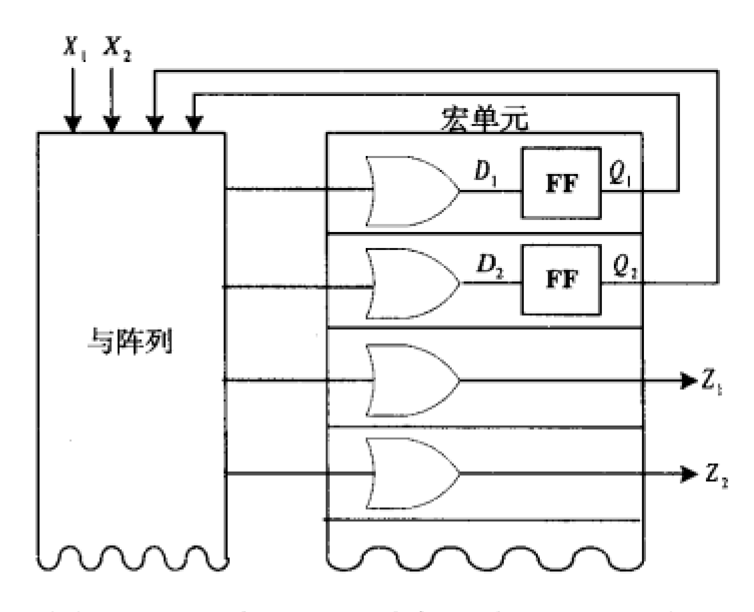


图 3.23　一个 Mealy 时序电路的 CPLD 实现

用 CPLD 实现基于累加器的并行加法器

下面我们要用 CPLD 实现一个基于累加器的加法器，如图 3.24 所示。在累加器寄存器中，每一位都需要一个触发器，而每一位都要生成一个和和进位位。

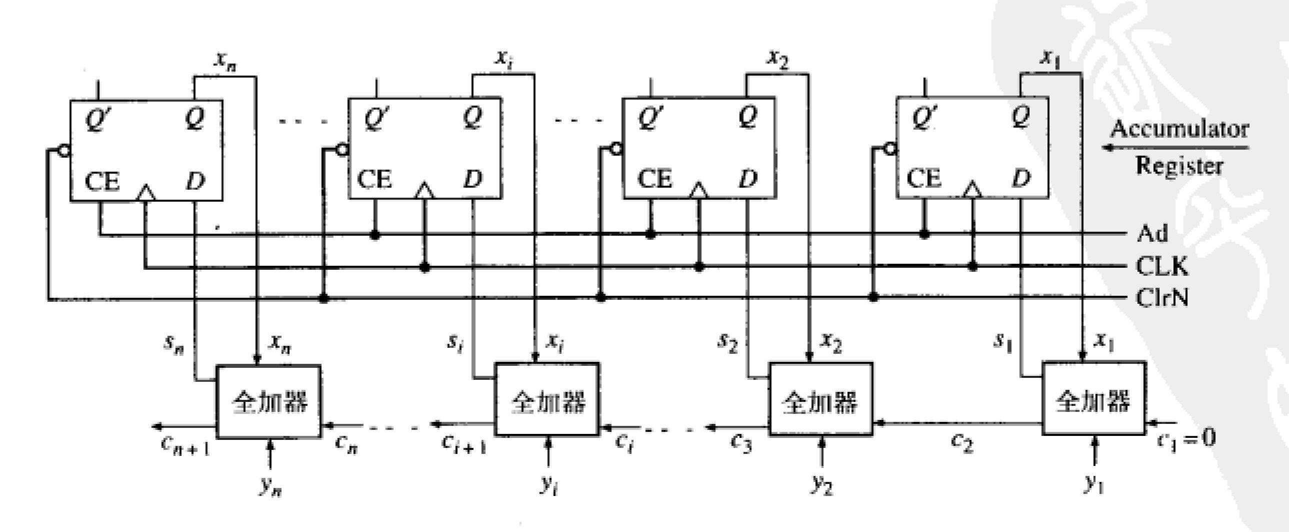


图 3.24　基于有累加器的 N 位并行加法器

图 3.25 说明了在一个 CPLD 中，怎样用一个累加器实现三位并行加法器。加法器的每一位都需要两个宏单元。一个宏单元用于实现求和运算和累加器的一个触发器，而另一个宏单元则用来计算进位，并把它反馈到与阵列。Ad 信号可以通过与门（没有画出）连接到每个触发器的使能输入端（CE）。加法器的每一位需要 8 个乘积项（4 个用于求和，3 个用于进位计算，一个用于 CE）。对于每个累加器的触发器有

$$D_i = X_i^+ = S_i = X_i \oplus Y_i \oplus C_i$$

如果该触发器编程为 T 触发器，则求和逻辑表达式可以化简。对于每个累加器的触发器有

$$X_i^+ = X_i \oplus Y_i \oplus C_i$$

因此，T 触发器的输入为

$$T_i = X_i^+ \oplus X_i = Y_i \oplus C_i$$

可以把加信号可以和 T 触发器的输入 T_i 与起来，这样只有 $Ad = 1$ 时，触发器状态才发生改变：

$$T_i = Ad(Y_i \oplus C_i) = Ad\, Y_i C_i' + Ad\, Y_i' C_i$$

进位的逻辑表达式为

$$C_{i+1} = X_i Y_i + X_i C_i + Y_i C_i$$

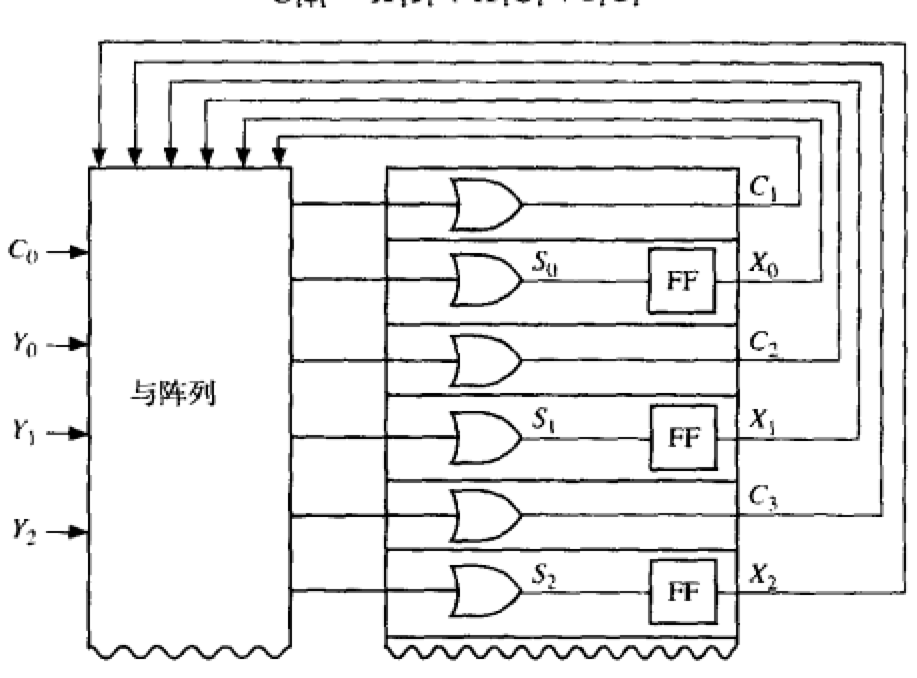


图 3.25　基于累加器的 N 位并行加法器的 CPLD 实现

3.4 现场可编程门阵列

这一节我们介绍现场可编程门阵列（FPGA）。FPGA 是在一片 IC 上集成了一个由相同的逻辑单元组成的阵列和可编程连接模块。用户可以通过对每一个逻辑单元及其连接进行编程来实现函数。FPGA 根本上改变了原型和设计方法。FPGA 可重构编程的灵活性增强了设计的过程。Xilinx 公司使用静态 RAM（SRAM）存储单元设计了一种新的可编程逻辑块，并在 1985 年推出 XC2000 系列产品。尽管当时有很多不同种类的可编程逻辑器件可用，但是人们还是很快地接受了这种全新的、强大的技术。现在市场上有很多可选的 FPGA 产品。Xilinx，Altera，Lattice Semiconductor，Actel，Cypress，QuickLogic 和 Atmel 等公司都在设计和出售 FPGA。

与传统门阵列和掩模可编程门阵列（MPGA）相比，FPGA 具有很多的优点。传统门阵列可以用来设计任何电路，但是只能在工厂中一次性编程，而且还需要针对该电路的特定的掩模。因

此，设计一块基于门阵列的 IC 需要几个月。FPGA 是标准通用器件。如果设计人员用 FPGA 代替 MPGA，则设计时间将由几个月缩短至几小时，并且使设计的反复修改更加容易，显著地缩短了从设计到投放市场的时间。FPGA 使设计纠错过程更加简单，从而减少了错误修改和设计指标变更的花费。原型费用也减少了，小容量 FPGA 比 MPGA 要便宜。

FPGA 也存在缺点。FPGA 的密度比传统的门阵列（MPGA）要低。在 FPGA 中，很多资源仅仅为了保证其可编程性。MPGA 的性能要比 FPGA 好。在 FPGA 中，每个可编程的点都有电阻和电容。电阻和电容的使用减慢了信号的传输速度，所以 FPGA 的速度比传统门阵列低。而且，FPGA 中互联延迟是不可预测的。由于 PAL 和 GAL 等 PLD 器件结构简单，价钱便宜。所以 CPLD 比 FPGA 速度快，而且价钱便宜。但是 PAL 和 CPLD 的可编程性较弱。与 FPGA 相比，CPLD 的优点在于其价钱便宜和时序的可预测性。

表 3.7 中列出了多个商用 FPGA。我们可以看到发现有些芯片包含的逻辑相当于 500 万门。有些 FPGA 的容量则由查找表（LUT）个数来给出。由于 FPGA 的存储量很大，所以可以使用单片 FPGA 实现和构建大型系统。本节中，我们将介绍 FPGA 的基本构成，FPGA 的设计实例将在第 6 章中介绍。

表 3.7　商用 FPGA 示例

生产商	FPGA 产品	容量（门/LUT）
Xilinx	Spartan-Ⅱ	15 ~ 200k
	Spartan-ⅡE	50 ~ 600k
	Spartan-3	50k ~ 5M
	Virtex-5	19 200 ~ 207 360 LUTs
	Vittex	57 906 ~ 1 124 022
	Virtex-E	71 693 ~ 4 074 387
	Virtex-Ⅱ	40k ~ 8M
Altera	ACEX 1k	56 ~ 257k
	APEX Ⅱ	1.9 ~ 5.25M
	FLEX 10k	10 ~ 50k
	Stratix/Stratix Ⅱ	10 570 ~ 132 540 个逻辑单元
Lattice Semiconductor	LatticeECP2	6 ~ 68k LUT
	Lattice SC	15.2 ~ 115.2k LUT
	ispXPGA	139k ~ 1.25M
	MachXO	256 ~ 2280 LUTs
	LatticeECP	6.1 ~ 32.8k LUTs
Actel	Axcelerator	125K ~ 2M
	eX	3 ~ 12k
	ProASIC3	30k ~ 3M
	MX	3 ~ 54k
Quick Logic	Eclipse/EclipsePlus	248 ~ 662k
	Quick RAM	45 ~ 176k
	pASIC 3	5 ~ 75k
Atmel	AT40K	5 ~ 40k
	AT40KAL	5 ~ 50k

3.4.1 FPGA 的结构

图 3.26 给出了一个典型的 FPGA 的基本布局。FPGA 内部基本上包含三个可编程单元：

可编程逻辑模块
可编程输入/输出模块
可编程布线资源

可编程逻辑模块阵列分布在 FPGA 内，这些逻辑模块阵列是被输入/输出（I/O）接口模块所包围的。I/O 模块分布在芯片的外围，它们把逻辑信号连接到 FPGA 的引脚上。逻辑模块之间的空间用于在模块间进行布线连接。

FPGA 的“现场”可编程能力取决于可重构单元，它们可以编程或者可编程重构。如上所述，FPGA 有三个主要可编程单元：逻辑模块、互连和输入/输出模块。可编程逻辑模块由多路选择器、查找（LUT）和与-或阵列（或者为与非-与非阵列）构成。所谓“编程”，是指改变多路选择器的输入或控制信号，或者改变查找（LUT）的内容，或者选中/不选中特定的与-或模块中的特定门。对于可编程互连来说，“编程”即为连通或者断开某一连接。当我们要使芯片中的各种模块之间相互联接，或者想要特定的 I/O 引脚同特定的逻辑模块相连时，就需要对互联模块进行编程。可编程 I/O 模块则指某一模块的输入、输出或者是双向的可编程性。通常 I/O 模块也可以“编程”调整缓冲器的特性，比如反相/不反相、三态、有源上拉，或者甚至调节转换率（Slew Rate），即改变某一引脚上信号的变化速率。

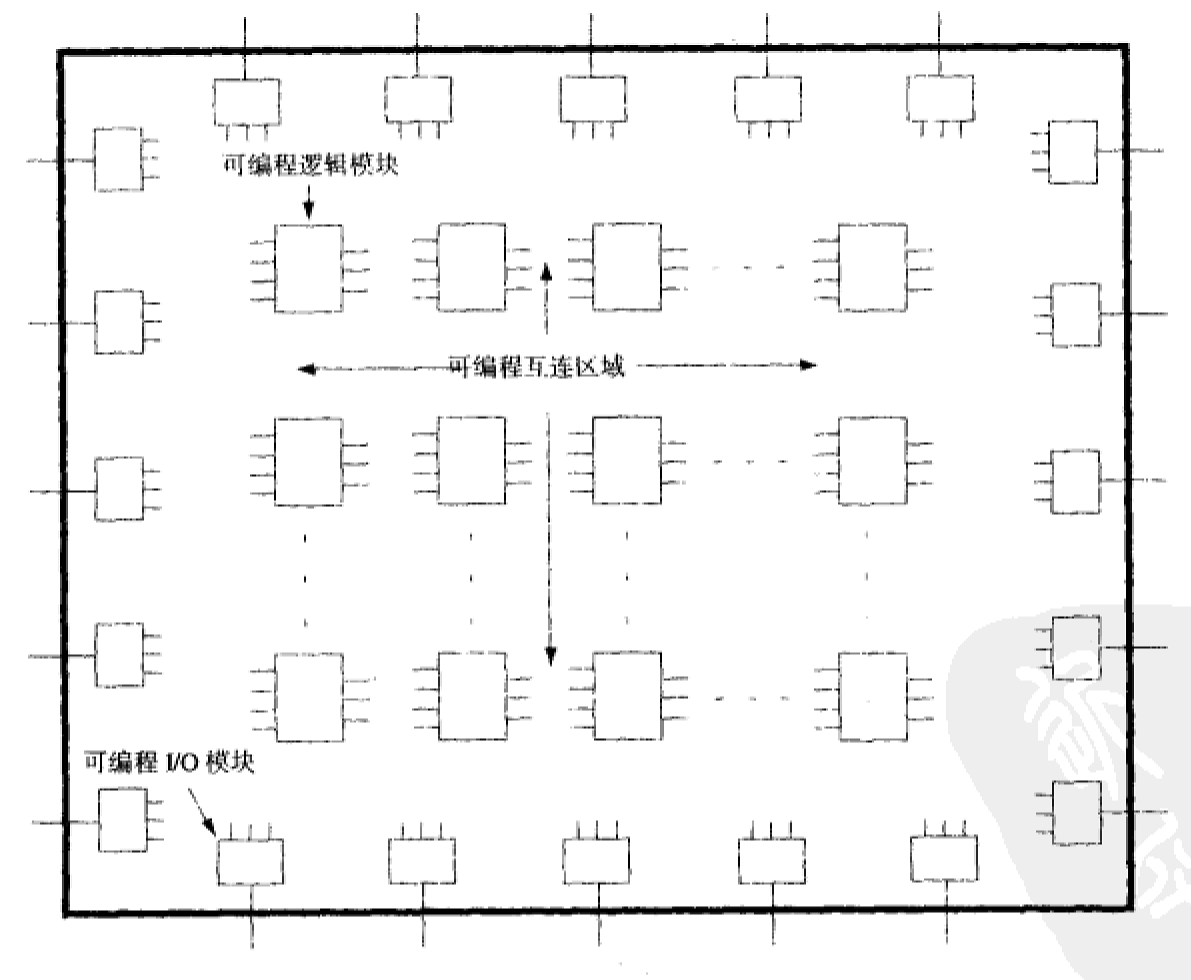


图 3.26 典型 FPGA 的基本结构（Layout）

灵活的通用互联模块是 FPGA 区别于 CPLD 的主要特点。CPLD 中的互联是受限的，而 FPGA 中的通用互联是很灵活的。但 FPGA 也有速度慢的缺点。当连接芯片的两个部分的时候，要经过多个可编程互连点，所以其信号延迟很大而且不可预测。

虽然图 3.26 用于表示 FPGA 的一般结构，但不是所有的 FPGA 都具有如图 3.26 所示的结构。商用 FPGA 有各种不同的结构。FPGA 的结构布局和组成取决于各个逻辑模块和互连资源在 FPGA 内部的布局方法和拓扑结构。图 3.26 所示的结构一般称为对称阵列的结构。20 世纪 80 年代末至今出现了各种 FPGA，它们可以分为 4 种不同结构：

矩阵（对称阵列）型
横向型
从属 PLD 型
门海型

图 3.27 说明了这几种结构。

矩阵（对称阵列）型

这种 FPGA 的逻辑模块的排列是一个矩阵型，如图 3.27(a)所示。大多数 Xilinx 生产的 FPGA 都属于这种结构，它的逻辑模块具有很强的功能，能够实现 4 变量或更多变量逻辑函数表达式。这种结构的 FPGA，较小的芯片上通常有 8×8 的阵列，较大的芯片上通常有 100×100 的阵列。布线资源分布在逻辑模块之间。由于只在水平和垂直方向上有布线资源，所以此类 FPGA 的布线被称为二维通道布线。

横向型

这种结构来自受传统门阵列结构。这种 FPGA 的逻辑模块是按行排列的，如图 3.27(b)所示。因此，布线资源也是按照行排列的。分布在行与行之间的布线资源可以用来连接各个逻辑模块。传统的掩模可编程逻辑门阵列也有类似的结构。由于这种结构的布线资源都分布在逻辑模块的行与行之间，所以其布线被称为一维通道布线。有些 Actel 生产的 FPGA 采用这种结构。

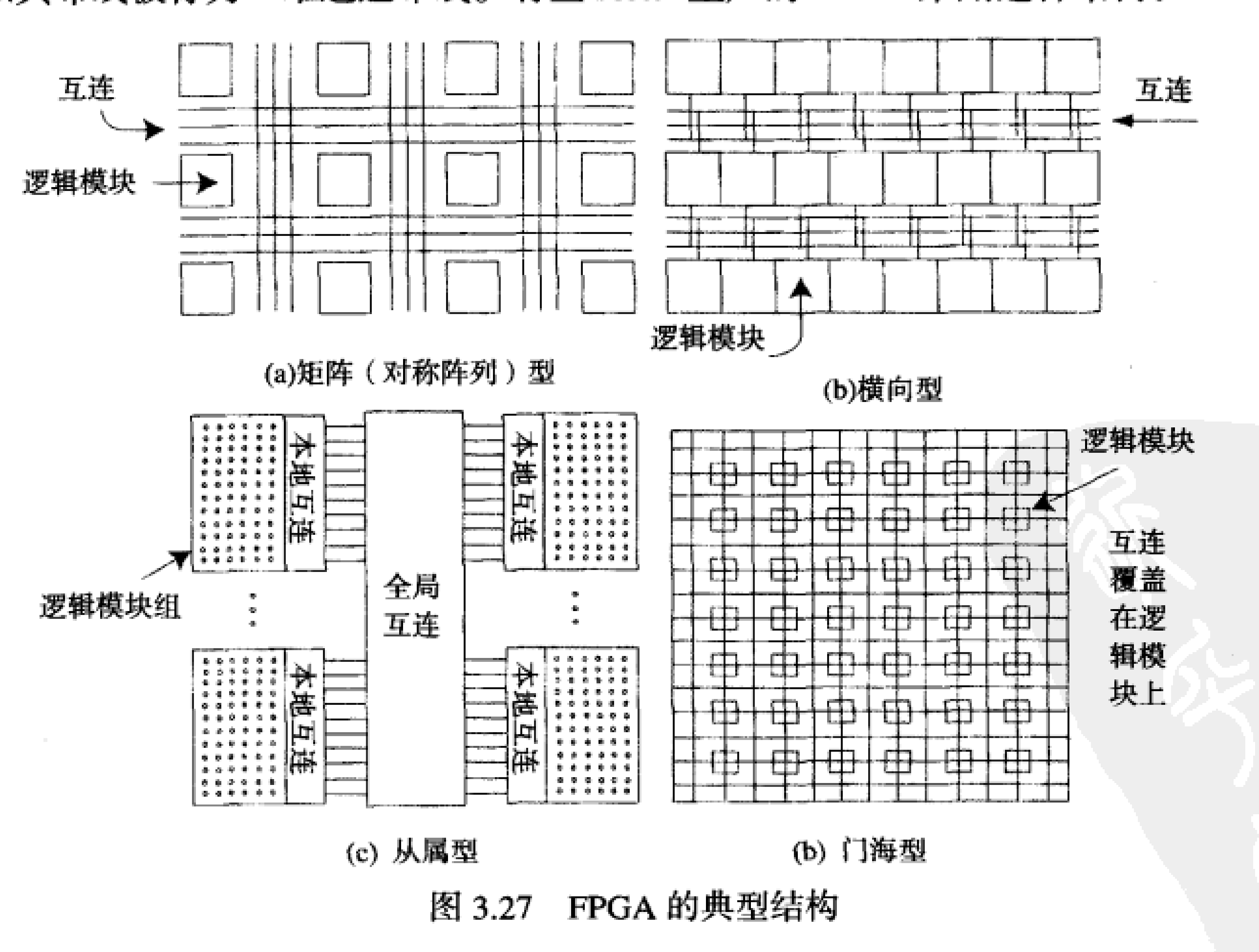


图 3.27 FPGA 的典型结构

从属型

有些 FPGA 中，逻辑模块的单元被分成几组分别由各自的本地互联接起来形成一个个模块，几组这样的模块由另一层面的互联链接起来。例如，Altera 生产的 APEX20 和 APEX II 型 FPGA 中，大概有 10 个左右的逻辑单元被连接起来形成一个逻辑阵列模块（Altera 公司称之为 LAB），然后多个 LAB 连接起来构成一个 MEGALAB。因此，此类 FPGA 的结构是具有从属关系的。此种 FPGA 包含逻辑模块集群和互联逻辑资源。全局互联网络用来实现逻辑模块集群间的相互连接。

门海型

FPGA 的另一种组织逻辑模块和互联的结构是门海型结构。通常，FPGA 芯片上都包含大量的门，并且互联在门海中从一处移到另一处，如图 3.27(c)所示。Plessey 公司在 20 世纪 90 年代中期发明了这种结构的 FPGA。此种 FPGA 的基本单元为与非门。作为一个术语，门海（sea of gates）更为常用，它也称为**单元海**（**sea of cells**）或**瓦片海**（**sea of tiles**）。它们都用来形容此种 FPGA 含有大量的逻辑单元，并且这些逻辑单元排列整齐。Actel Fusion 生产过这种类型的 FPGA，其中每个瓦片都可以配置为一个 3 输入逻辑函数或一个触发器/锁存器。

3.4.2 FPGA 编程技术

FPGA 中分布着大量的逻辑模块，这些逻辑模块之间为可编程互联。对这些逻辑模块的编程是指该模块“编程”或者“配置”实现任何所需电路。逻辑模块之间的互连也是可以编程的。

有几种技术可以用来确保 FPGA 中的互连是可编程的。我们所说的编程技术是指那些能达到 FPGA 可编程性的技术。在有些器件中，可以通过改变静态 RAM 单元的内容进而实现器件的可重构性。对于另外一些器件，其可重构性是通过使用 Flash 单元来实现的。还有些器件是通过金属熔丝连线实现的。一般 FPGA 采用的如下几种编程技术：

静态 RAM（SRAM）编程技术
EPROM/EEPROM/flash 编程技术
反熔丝编程技术

SRAM 编程技术

SRAM 编程技术是指通过存储在静态 RAM（SRAM）单元中的数据来产生可重构性。通过设置 SRAM 中存储的数据，逻辑模块、I/O 模块和互联可以实现其可编程性。可重构逻辑模块可以简单地通过查找表（LUT）实现，这与 3.2.1 节中所用的基于 ROM 的方法相同。16 个 SRAM 单元可以用来实现任意 4 变量逻辑函数。互连的可编程性也可以通过 SRAM 来实现。该技术的关键思想是用传输晶体管作为开关，用 SRAM 中的数据来控制这一开关。如图 3.28(a)所示，SRAM 单元与传输晶体管的栅极相连。当 SRAM 单元内容为 0 时，传输晶体管就被阻断，此时 A 和 B 两点断开。当 SRAM 单元内容为 1 时，传输晶体管连通，此时 A 和 B 两点形成通路。通过使用多路选择器，SRAM 单元中存储的比特可以用来构建布线矩阵，如图 3.28(b)所示。通过改变图 3.28(b)中 SRAM 的内容，可以使设计者改变与 X 点相连的输入。SRAM 中存储的内容可以决定 LUT 的功能和互联，所以我们把这个数据比特称为**配置比特**（Configuration Bits）。

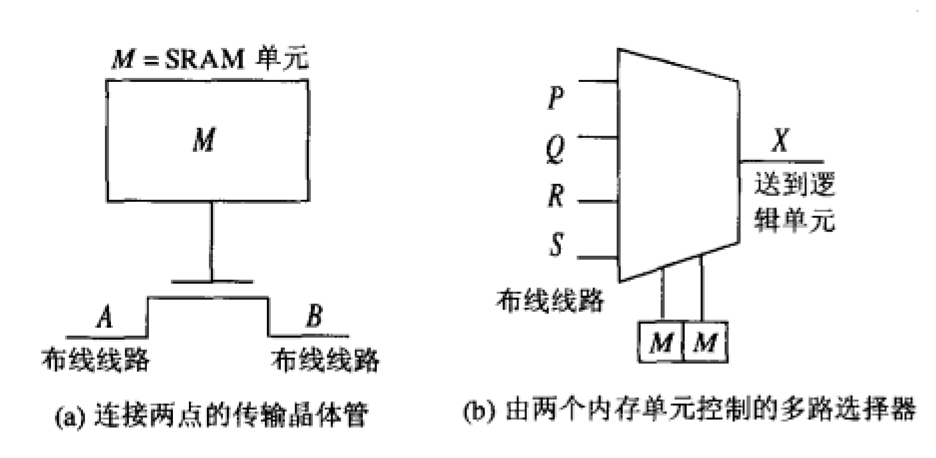


图 3.28　静态 RAM 可编程技术的布线

一个 SRAM 单元通常需要 6 个晶体管实现，如图 3.29 所示。4 个交叉耦合的晶体管构成一个锁存器，另外两个晶体管用来控制传输到锁存器中的数据比特。当 Word Line 为高电平时，Bit Line 上的数值将被锁存，这就是写操作。在进行读操作时，要使 Bit Line 提前充电，Bit Line 变为逻辑 1，然后把 Word Line 设为高电平，随后存储单元的内容就出现在 Bit Line 上。有些 SRAM 单元只用 5 个晶体管就能实现。使用静态 RAM 的一个好处是它的的易失性，为此需要反复更新其内容，这就为其原型和开发阶段提供了灵活性。另一个优点是，制造 SRAM 单元的步骤与制造逻辑单元的步骤没有什么不同。SRAM 编程技术的主要缺点为：对于每个 SRAM 单元，我们都需要使用 5 或 6 个晶体管，这带来了制造费用增加。例如，如果 1 块 FPGA 芯片含有 100 万个可编程点，则要使用 500 万或 600 万晶体管来实现其可编程性。

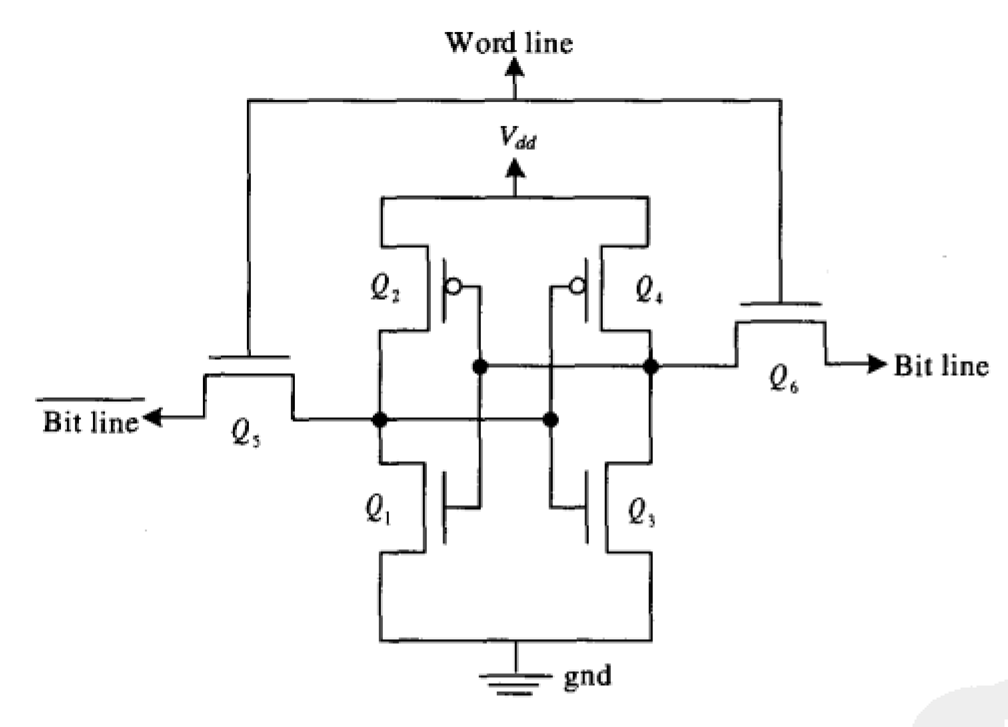


图 3.29　典型的 6 晶体管 SRAM 单元

如果最后成品阶段使用 FPGA，则易失性却成为 FPGA 的一个缺点。因此，在使用 SRAM FPGA 时，应使用非易失器件（如 EPROM）永久存储配置比特。我们是把 EPROM 作为引导 ROM 来使用的，当通电后，EPROM 中的内容就传输到 SRAM 中去。

Xilinx 的 FPGA 首先使用 SRAM 编程技术。实际上，正是因为 SRAM FPGA 的灵活性和可重构性使得它得到了广泛的应用。现在，许多公司都在他们的 FPGA 上使用 SRAM 编程技术。

EPROM/EEPROM 编程技术

在 EPROM/EEPROM 编程技术中，EPROM 单元用来控制可编程连接。假设用 EPROM/EEPROM 单元取代图 3.28 中的 SRAM 单元。构建一个 EPROM 单元（如图 3.30 所示）

需要使用一个双栅极晶体管（浮栅和控制栅）。上拉电阻把晶体管的漏极同电源（图中为 V_{dd}）连接起来。为了使晶体管开关断开，需要在浮栅上注入电荷，我们可以使用控制栅和晶体管漏极之间的高电压实现这一目的。这些注入电荷使晶体管的门限电压提高，此时晶体管断开。当用紫外线照射浮栅时，这些注入电荷将会被移走，晶体管的门限电压降低，因此晶体管导通。

EPROM 的速度比 SRAM 要慢，因此基于 SRAM 的 FPGA 器件编程速度更快。不仅如此，与 SRAM 相比，EPROM 需要更多的处理步骤。基于 EPROM 的开关具有很高的导通电阻和很高的静态功率消耗。EEPROM 与 EPROM 类似，但是移除注入电荷可以用电的方法实现。

Flash 存储器是 EEPROM 的一种。它的主要特性是通过一次操作可以擦除多个存储单元。同传统的 EPROM 一样，Flash 在浮栅晶体管中存储信息。浮栅由一个绝缘氧化物层隔离出来，因此任何放置其中的电子均被束缚。进行读操作时，我们要在控制栅处放置一个特定的电压。此时电流是否流动取决于单元的门限电压，而此门限电压是由浮栅中束缚电子的多少决定的。一些器件在电流出现或消失时存储信息。另一些器件可以感知流通的电流量，因此一个单元可以存储几个比特二进制信息。在擦除时，我们要在控制栅和源极之间加一个很大的电压，以使浮栅中的电子被拉出。Flash 是按段擦除的，而且一个模块中的所有单元可以同时被擦除。

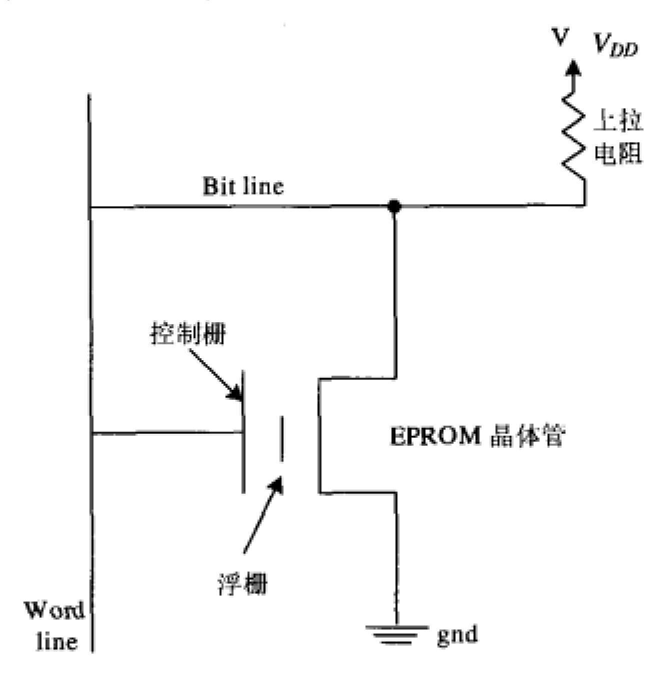


图 3.30　EPROM 可编程技术

反熔丝编程技术

在一些 FPGA 中，采用所谓的“反熔丝”来实现可编程连接。熔丝线在通过电流过大时会自动断开，而反熔丝编程单元，则加上高电压后，从高阻状态（断开）变为低阻状态（连通）。反熔丝通常用位于 N + 扩散层和多晶硅层之间的电解质层构建，或者使用金属层之间的非晶硅构建。通常反熔丝是断开的，但是被编程后将建立永久连接。此过程是不可逆的，因此反熔丝 FPGA 只可以编程一次。对反熔丝编程需要加高电压，使电流值超过普通电流值。为了使编程可以正常进行，所以我们需要使用特殊的编程晶体管（比普通晶体管大）。现存很多不同的反熔丝技术，通路反熔丝技术是其中较流行的一种。

由于可编程开关很小，所以反熔丝技术具有占地面积小的优点，而且基于反熔丝技术的互联的速度比基于 SRAM 和基于 FFPROM 的开关要快。当然，反熔丝技术也具有不可再次编程的缺点。由于它建立的连接是永久性的，所以当设计出现错误或要进行修改时，只能另外使用一块新的芯片。

各种编程技术比较

表 3.8 中比较了 FPGA 各种主要编程技术的特点。如表 3.8 所示，只有 SRAM 和 EEPROM 编程技术允许在线编程。在线编程是指在对 FPGA 进行重新编程时，不用把它从电路中取出。传统基于 EPROM 的器件不支持在线编程，但是 EEPROM/flash 技术支持在线重新编程。

SRAM FPGA 存在一些缺点：占地面积大，延迟大，具有易失性等。但是，由于它具有在线可编程特性和快速编程能力，所以得到了广泛的应用。SRAM FPGA 比其他类型的 FPGA 贵，这是由于它的每个编程点需要使用 6 个晶体管。这些额外的硬件对可重构编程性有很大的好处，但是不利于用 FPGA 实现实际电路。我们对基于 EEPROM/flash 的 FPGA 同 SRAM FPGA 在各方面进行了比较，但是前者的速度没有后者快。

表 3.8　主要 FPGA 可编程技术的特性

可编程技术	易变性	可编程性	面积	阻值	容量
SRAM	易变	在线可重构编程	大	中到高	高
EPROM	非易变	离线可重构编程	小	高	高
EEPROM	非易变	在线可重构编程	中到大	高	高
反熔丝	非易变	不可重复编程	小	小	小

3.4.3　可编程逻辑模块的结构

以前 FPGA 使用过各种不同的可编程逻辑模块作为其基本组成模块。本节中，我们对商用 FPGA 的一般典型组成模块进行介绍。

FPGA 中作为基本组成的逻辑模块不是一成不变的。例如，有些 FPGA 使用基于 LUT 的逻辑模块，另一些 FPGA 则使用多路选择器和逻辑门作为基本的逻辑模块，还有一些 FPGA 的逻辑模块只由晶体管对组成（如交叉点 FPGA）。早期 Altera 的 FPGA 其逻辑组成模块是 PLD 模块。还有一些 FPGA 用与非门作为基本逻辑模块。

逻辑模块的结构和大小也是变化的。有些 FPGA 使用大型基本模块，这些模块可以实现大型逻辑函数（多个 5 变量或 4 变量表达式），并且每个基本模块中都含有多个触发器。与之相对，有些 FPGA 的组成模块只能实现 3 变量逻辑函数且每个模块只有一个触发器。有些 FPGA 允许用户选择输出类型（带锁存器，不带锁存器，或二者均有），有些 FPGA 允许用户控制内部的触发器，还有一些 FPGA 是时钟上升沿/下降沿触发的，其触发器含有置位/复位输入等。不同的 FPGA 生产厂商使用不同的名字来表示各自的逻辑模块。Xilinx 生产的可编程逻辑模块称为**可配置逻辑模块**（**CLB**）；Altera 把它们的基本模块称为**逻辑单元**（**LE**），并把 8 或 10 个 LE 组合称为**逻辑阵列模块**（**LAB**），Actel Fusion 的基本模块称为 **VersaTile**。

基于查找表（LUT）的可编程逻辑模块

许多基于 LUT 的 FPGA 使用 4 变量 LUT 和一个触发器作为基本单元，然后把多个基本单元按一定的拓扑结构结合起来。观察图 3.31，该可编程逻辑模块含有两个 4 变量查找表（通常简写为 **LUT4**）和两个触发器。由于 LUT4 可以生成任意 6 变量逻辑函数，所以它也称为 6 变量逻辑函数生成器，那么两个 LUT4 就可以生成两个任意 4 变量逻辑函数。*X*-逻辑函数生成器的输入为 X_1, X_2, X_3 和 X_4，*Y*-函数生成器的输入为 Y_1, Y_2, Y_3 和 Y_4。这些函数的输出可以是组合逻辑的或含有锁存器。该逻辑模块中有两个 D 触发器，并且这些 D 触发器具有多样性，因为它含有时钟使能端、直接置位和复位端。多路选择器在组合逻辑输出和锁存器的锁存输出中选择一个作为最后的输

出。图中标识为 M 的模块是存储器单元，它生成选择信号控制多路选择器。早期 Xilinx 生产的 FPGA——XC3000 具有类似结构的基本模块。

假设一个 FPGA 具有如图 3.31 的基本单元，下面介绍如何用它实现逻辑函数 $F_1 = A'B'C + A'BC' + AB$。由于该逻辑函数含有 3 个变量，所以 4 输入 LUT 足以实现该逻辑函数。图 3.32 中的高亮路径说明要使用 X 函数生成器。假设 X_1 是 LUT 的最低有效位，X_4 是 LUT 的最高有效位。由于函数 F_1 只使用 3 个变量，所以输入 X_4 不使用。我们对该函数构建真值表，从而可以获得 LUT 的内容。

要实现逻辑函数 F_1，则 LUT 的内容为 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1。LUT 的前 8 位表示真值表中逻辑函数 F_1 的输出。由于输入 X_4 没有接地，所以 LUT 的前 8 位比特数据要重复使用一次，以备没有使用的 X_4 可能变为逻辑 1。由于逻辑函数是存储在 LUT 表中，所以逻辑函数的乘积项个数并不重要，可以不必对该函数表达式进行化简，真正重要的是变量的个数。

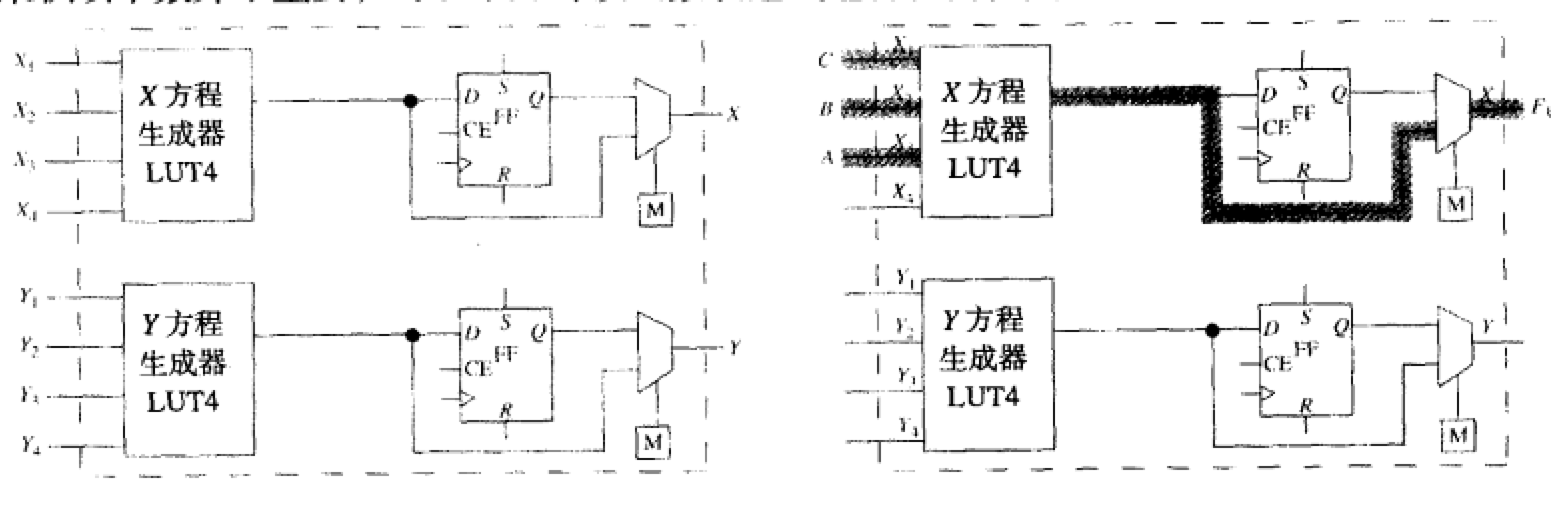


图 3.31 基于查找表（LUT）的可编程逻辑模块

图 3.32 方程 F_1 的高亮路径

很多商用 FPGA 都使用 LUT，如 Xilinx 公司生产的 Spartan/Virtex，Altera 公司生产的 Cyclone II /APEX II，QuickLogic 公司生产的 Eclipse/PolarPro，Lattice Semiconductor 公司生产的 ECP。这些 FPGA 中有很多把 2 个或 2 个以上的 LUT 按不同的拓扑结构放到一个模块中。有些 FPGA 还用多路选择器配合 LUT 使用。

基于多路选择器和门电路的逻辑模块

有些 FPGA 使用多路选择器作为基本组成模块。众所周知，只使用多路选择器就可以实现任何组合逻辑表达式。举例来说，一个 4 选 1 多路选择器可以实现任何 2 输入表达式。如果多路选择器的输入可以是反相输入，则一个 4 选 1 多路选择器可以实现任何 3 输入表达式。基于多路选择器的基本模块如图 3.33 所示。早期 Actel 公司生产的 ACT Ⅰ 和 ACT Ⅱ 型号的 FPGA 就采用类似结构。

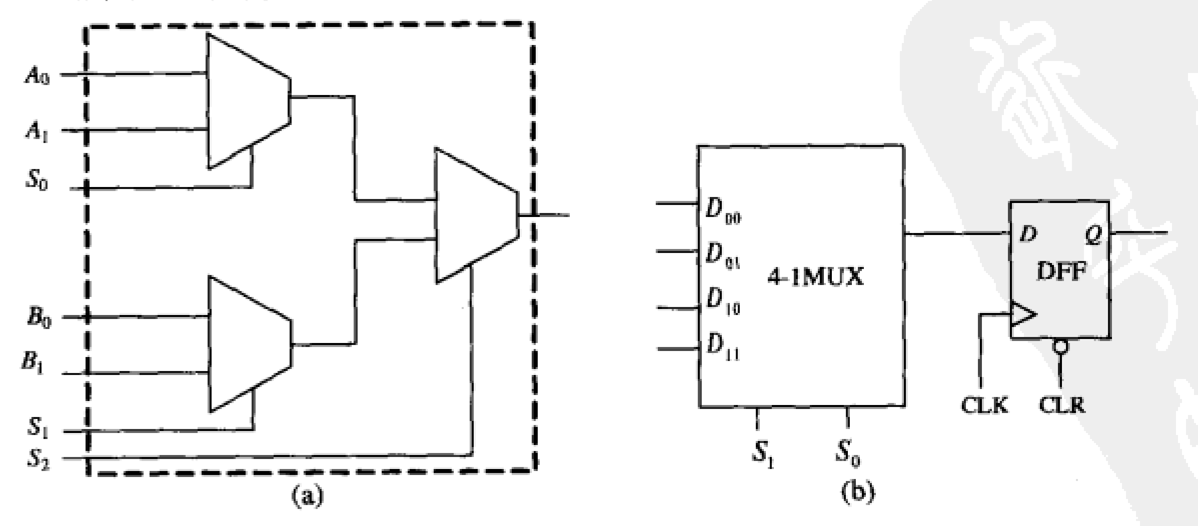


图 3.33 FPGA 中基于多路选择器的逻辑模块

假设一个 FPGA 其内部基本逻辑模块为 4 选 1 多路选择器。下面介绍如何使用该 FPGA 实现逻辑函数 $F_1 = A'B'C + A'BC' + AB$。3 个输入变量中的 2 个连接到多路选择器的选择输入端上，随后为了要实现该函数，我们要为多路选择器数据输入端提供适当的信号。为了对这些输入进行推导，我们先建立如下真值表：

A B C	F	Mux 输入{0, 1, C, C'}
0 0 0	0	C
0 0 1	1	
0 1 0	1	C'
0 1 1	0	
1 0 0	0	0
1 0 1	0	
1 1 0	1	1
1 1 1	1	

假设 A 和 B 连接到多路选择器的选择输入端，第 3 个变量 C 连接到多路选择器的数据输入端。当多路选择器的输入为{C, C', 0, 1}，则此多路选择器可以表示任意 3 变量逻辑函数。下面观察其真值表，我们发现当 $AB = 00$ 时，$F = C$（真值表头 2 行）；当 $AB = 01$ 时，$F = C'$（真值表第 3，4 行）；当 $AB = 10$ 时，$F = 0$ 且与 C 的值无关；当 $AB = 11$ 时，$F = 1$。真值表中最后一列表示多路选择器的理想输入。因此，如图 3.34 所示的 4 选 1 多路选择器可以实现表达式 F_1。

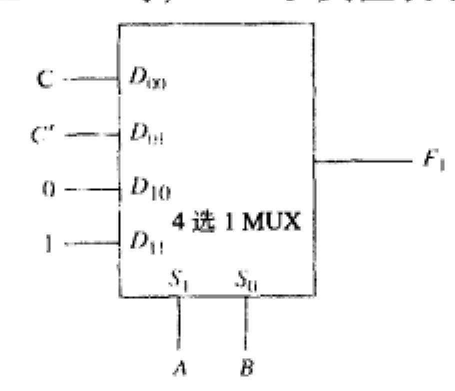


图 3.34 用多路选择器实现方程 F_1

到此为止，我们简单介绍了 FPGA 的基本结构、逻辑模块类型和所用编程技术。表 3.9 中列出了一些商用 FPGA 的基本结构、逻辑模块类型和所用编程技术。基于 LUT 的 FPGA 是很常见的，尤其在 Xilinx 和 Altera 公司生产的 FPGA 中更为常见。Actel 公司生产的 FPGA 是基于多路选择器的。尽管基于 SRAM 编程技术的 FPGA 较贵，但是也很常用。

表 3.9 商用 FPGA 的结构，技术和逻辑模块类型

生 产 商	器件名称	基本结构	逻辑模块类型	可编程技术
Actel	ProASIC/ProASIC3/ProASIC^{plus}	海量片	多路选择器和基本门	SRAM
	SX/SXA/eX/MX	海量模块	多路选择器和基本门	反熔丝
	Accelerator	海量模块	多路选择器和基本门	SRAM
	Fusion	海量片	多路选择器和基本门	Flash , SRAM
Xlinx	Virtex	对称阵列	LUT	SRAM
	Spartan	对称阵列	LUT	SRAM
Atmel	AT40KAL	基于单元	多路选择器和基本门	SRAM
QuickLogic	Eclipse II	可变时钟	LUT	SRAM
	PolarPro	基于单元	LUT	SRAM
Altera	Cyclone II	基于二维行和列	LUT	SRAM
	Stratix II	基于二维行和列	LUT	SRAM
	APEX II	行和列，但是分层互联	LUT	SRAM

3.4.4 可编程互联

通用可编程互联是 FPGA 的一个重要组成单元，它按照一定的结构分布在可编程逻辑模块之间。对于商用 FPGA 来说，有很多种互联资源，每个生产商生产的不同种类的互连都有不同的名字。

对称阵列 FPGA 中的互联

本小节中，我们将介绍对称阵列 FPGA 中互联的基本单元。

通用互联：很多 FPGA 都使用开关矩阵，这些开关矩阵是由布线线路之间的互联链接起来的，如图 3.35(a)所示。典型的开关矩阵如图 3.35(b)所示，图中每个互联上都有一个开关。支持任意两线相连的开关矩阵是很昂贵的，通常芯片只支持局限于纵横互联，而且链接不可能同时存在。从图 3.35(b)中可以看出，通过使用开关组合，我们可以把开关一边的线同开关另一边的线连接起来。为了实现此种链接，开关矩阵的每个交叉点都必须支持 6 个可能的互联，这些互联如图 3.35(c)所示。

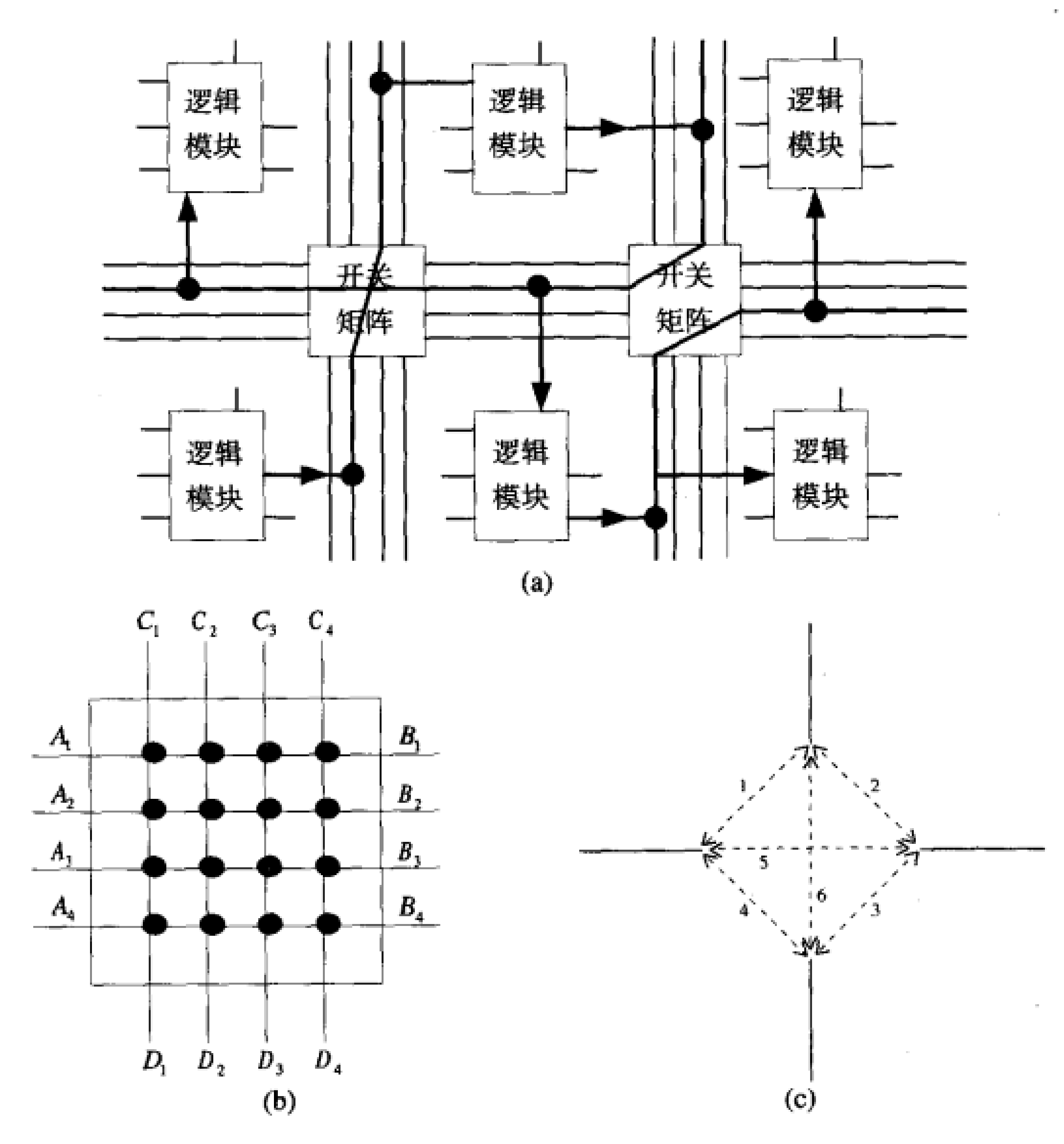


图 3.35　FPGA 中通用互联的布线矩阵

SRAM 单元、Flash 单元或者反熔丝连接可以用来控制开关的设置。FPGA 中分布于逻辑模块之间的开关矩阵使芯片上任意点之间的互联成为可能。但是，开关矩阵所占面积较大，而且延迟较长。如果一个信号从多个此种开关矩阵通过将会引起很大的信号延迟。同时，此延迟是可变的、不可预测的，延迟的长短完全取决于信号通过开关矩阵的个数。相对地，CPLD 中的互联资源是受限制的，CPLD 的互联可以带来较小的、可以预测的延迟。

直接互联：很多 FPGA 在相邻的逻辑模块间采用特殊的链接。这些互联速度很快，因为它们不必遍历布线阵列。很多 FPGA 提供上下左右 4 个相邻逻辑模块的直接互联。图 3.36 是直接互联的示例。在某些情况下，芯片可以提供对角相邻的 8 个逻辑模块之间的直接互联，如图 3.36(b)所示。直接互联不使用开关矩阵，而是使用专用开关，所以其延迟较小。上面几类直接互联应用于 Xilinx 公司生产的一些 FPGA 上。

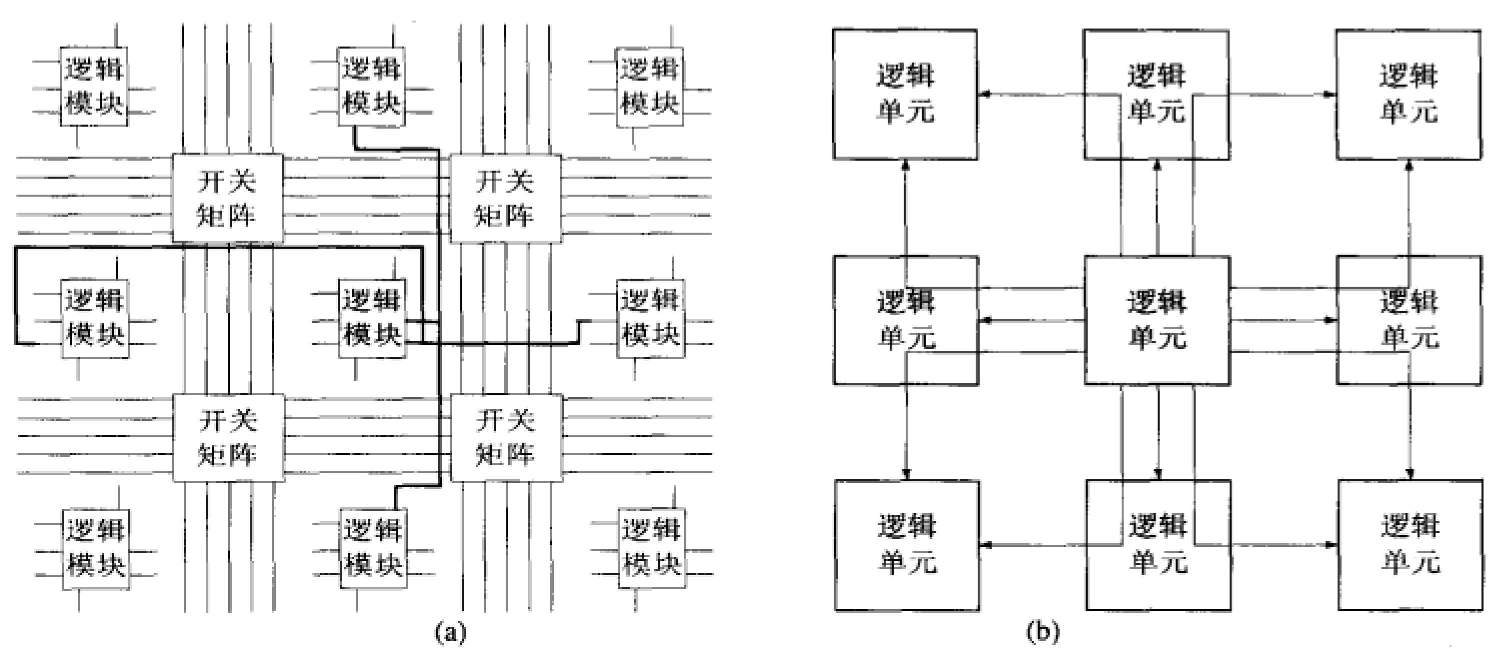


图 3.36　相邻逻辑单元间的互联

全局线：为了获得高扇出和低偏移的时钟失真，大多数的 FPGA 提供布线线路以提升器件的厚度（高度）。很多 FPGA 中在水平和垂直方向上提供有限（2 或 4）条此类全局线。图 3.37 中水平方向上的长线就是全局线。逻辑模块往往通过三态缓冲与全局线连接。

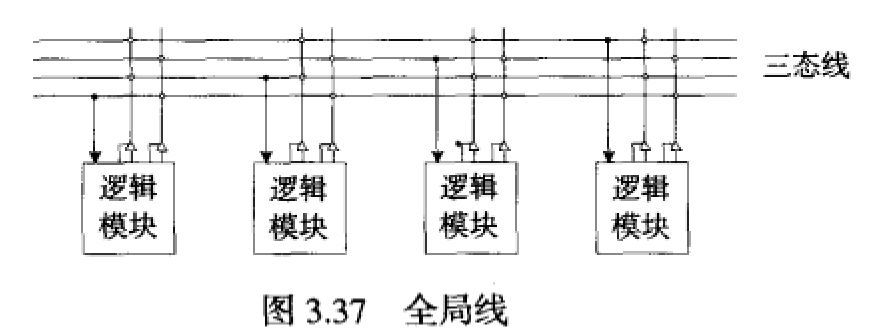


图 3.37　全局线

时钟偏移

现代 FPGA 芯片由几百万门组成。如果在芯片的各个部分都是时钟有效的，由于线路传播时钟信号存在延迟，所以时钟有效沿到达芯片内各部分的时间不同。实际时钟沿到达时间与理想时钟沿到达时间之差称为时钟偏移。包含现代微处理器在内的大型系统中，时钟偏移都是一个需要解决的问题。在大多数系统中，我们应仔细安排时钟电路的分布，把时钟偏移造成的影响最小化。现在 FPGA 使用特殊的时钟分布电路以获得足够长的时钟和较低的时钟偏移。

横向 FPGA 的互联

以前介绍的很多互联资源都是对称阵列器件的特性。这些对称阵列器件的逻辑模块是按二维阵列排列的。横向器件中，逻辑模块是按行排列的，并且各个逻辑模块通过开关通道连接起来。我们使用几个开关把信号从某行中的逻辑模块路由到芯片中其他行的逻辑模块。逻辑模块行与行之间的布线通道中含有开关阵列。此类 FPGA 中的布线资源同传统门阵列中的布线资源大体相同。

横向互联通道的结构可以分为两类：非分段布线和分段布线。为了更深入地了解不同种类的通道布线，下面我们举个例子（如图 3.38 所示）。非分段通道布线的结构如图 3.38(b)所示。图中水平方向有 3 行，在垂直方向上有多条线，在交叉点上有多个开关。尽管采用此种通道布线方式的 FPGA 基本上都使用反熔丝编程技术，但是其实对开关进行编程时可使用任意编程技术（SRAM, EPROM 或反熔丝编程技术）。通过对恰当的开关进行编程，我们可以得到想要的链接。我们通过对第 1 行第 1 列和第 1 行第 4 列上的开关进行编程，可以把 x 的两个端点连接起来。通常我们称

之为网 x。网 x 只是一条命名为 x 的线。通过对第 2 行第 2 列和第 2 行第 8 列上的开关进行编程，可以把 y 的两个端点连接起来，从而得到网 y。第 2 行被网 y 独占。这里我们可以看出此类互联资源的一个问题：即使对于一个很短的网，此类互联也要使用全长路径（如网 y 使用了整个第 2 行）。因此，此类互联所占面积很大。

为了解决因为使用全长路径而导致的占用面积过大的问题，我们使用分段路径。在分段路径中，每个路径都被分成了几段。如果第 1 行被分成两段，则我们可以用相同的路径构建另一个网。例如，网 x 和网 z 可以同时布线在第 1 行[如图 3.38(c)所示]。这就是分段路径布线的原理。分段路径布线可以使多个网具有同一路径，但是当需要使用长网时，必须在分段布线中使用段间开关。虽然这些开关在网中引入了更多的电阻和电容，但是总体来说分段布线降低了布线资源面积。

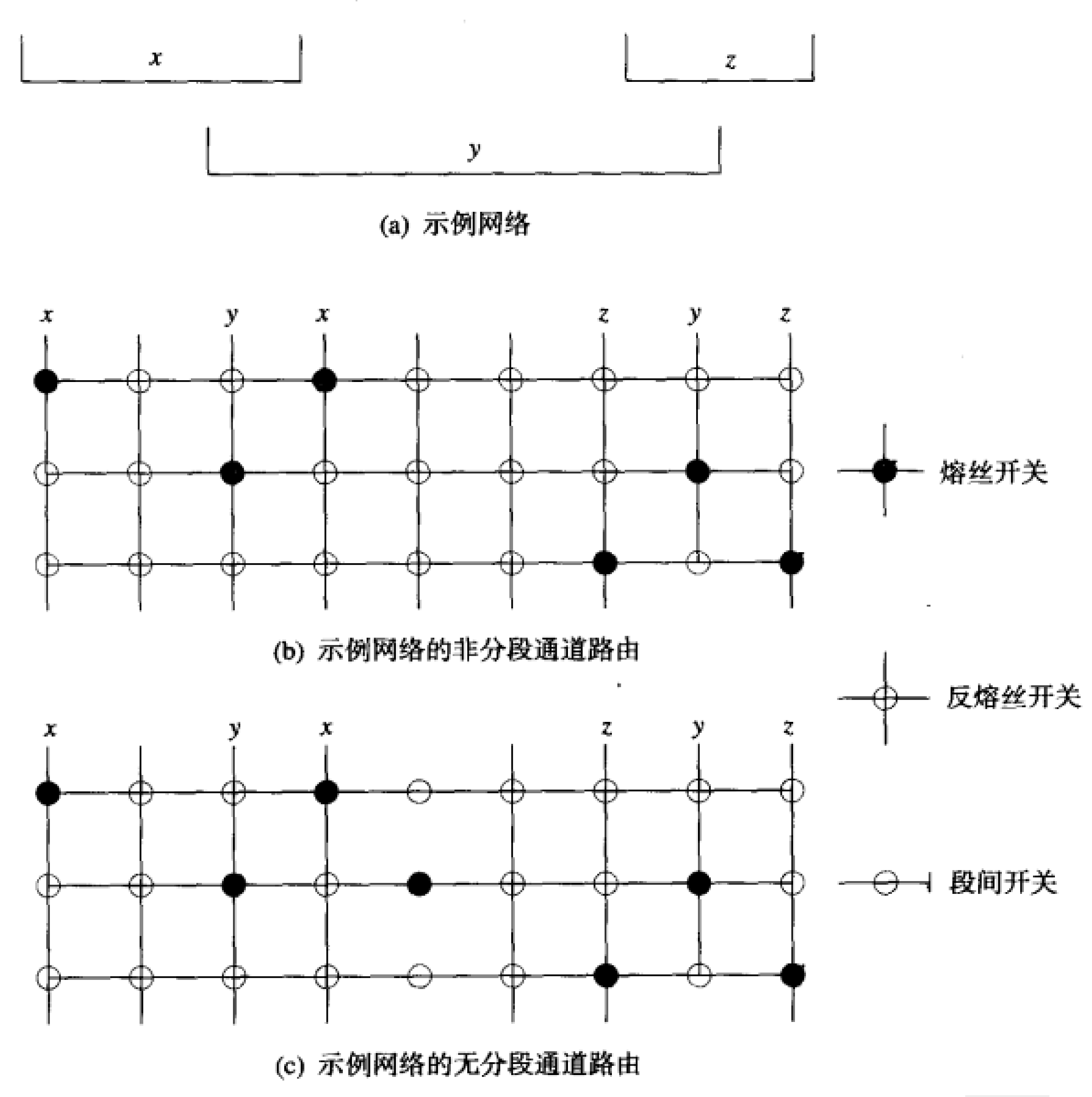


图 3.38　横向 FPGA 的典型布线资源

3.4.5　FPGA 中的可编程 I/O 模块

焊接在 FPGA 上的 I/O 连接在可编程输入/输出模块上。它使 FPGA 逻辑模块产生的信号以恰当的形式和格式更容易地传输到外部世界。现代 FPGA 中的 I/O 模块的管脚既可以做输出，也可以做输入，还可以是直接（组合）或锁存模式、三态缓冲模式等，它满足多种 I/O 标准。

图 3.39 展示了一个可设置输入输出模块（I/OB）。每个 I/OB 都有很多 I/O 选项，这些选项可以由设置存储单元（图 3.39 中用 M 表示）选择。I/O 管脚既可以为输入，也可以为输出。当作为输出时，需要使用三态缓冲器；当作为输入时，三态控制器必须设置三态缓冲器使其驱动输出管脚为高阻状态。

由于触发器的存在，所以输入和输出可以存储在 I/O 模块内部。当我们需要直接输入或输出时，触发器直接被通过，输入或输出不会被存储。许多 FPGA 的触发器输入可以设置成有效沿触发的 D 触发器，或者设置成锁存器。甚至当 I/O 管脚没有被使用时，I/O 触发器仍可以用来存储数据。

设置存储单元（图 3.39 中用 M 表示）可以控制多种相关 I/O 模块。如果需要的话，输出信号可以由 I/O 模块通过 XOR 门取反。当输出信号通过 XOR 门时，其取反与否取决于输出反相单元中设置比特的内容。三态反相设置比特可以生成一个高电平有效（或低电平有效）的三态控制信号。如果三态信号为 1，则 3-STATE INVERT 比特为 0（如果三态信号为 0，则三态反相比特为 1），输出缓冲器输出为高阻。还有一种情况下输出缓冲器一定是高阻状态：缓冲器把输出信号驱动到 I/O 管脚上，并把 I/O 管脚设置为输入状态，此时输出缓冲器一定为高阻。当一个外部信号到达 I/O 管脚后，它通过缓冲器并最终到达 D 触发器的输入端。这时，缓冲器输出一个直接输入信号到逻辑阵列中。或者此外部信号被存储在 D 触发器中，触发器对逻辑阵列发出一个 LATCHED IN 信号。

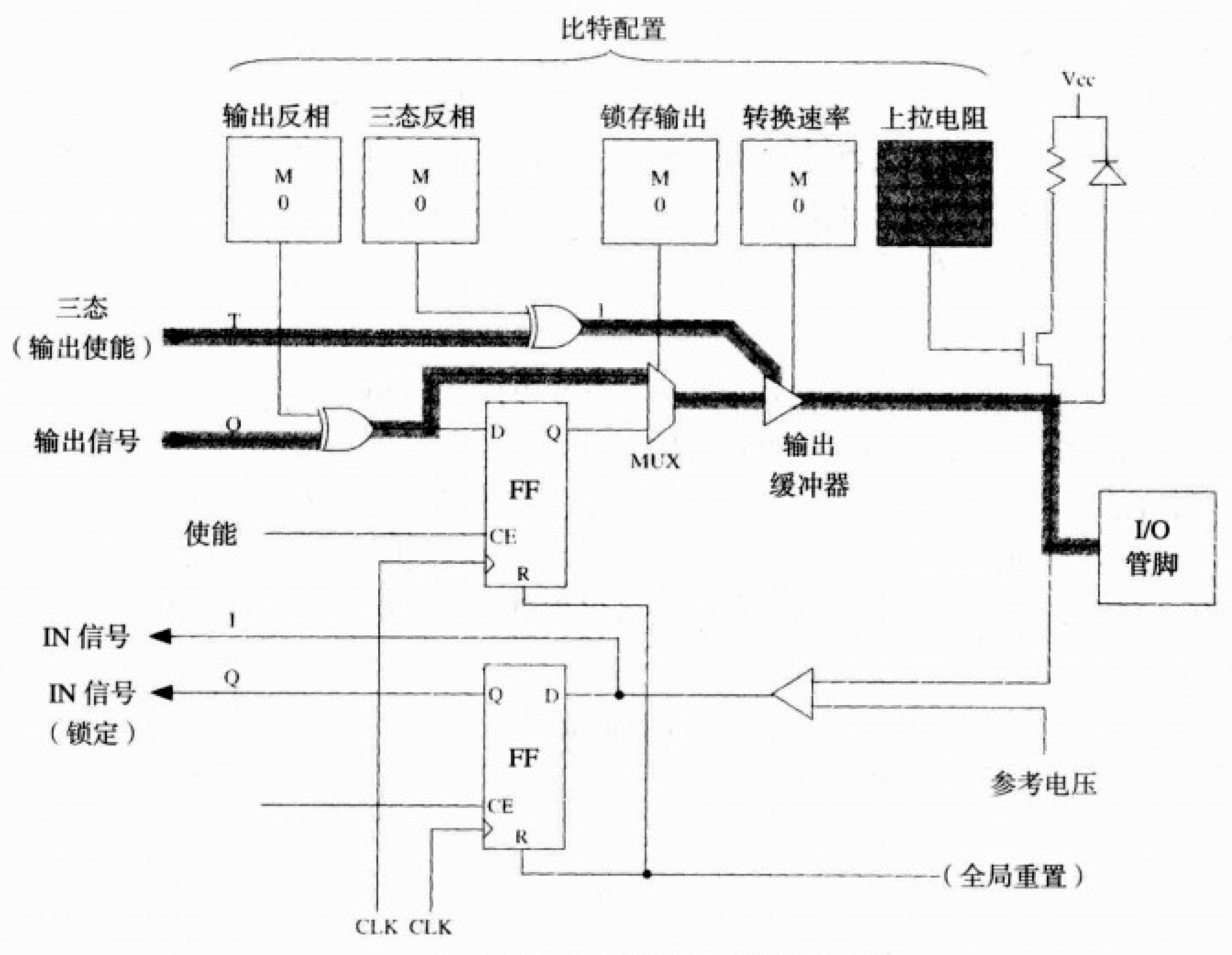


图 3.39　FPGA 的可编程 I/O 模块

锁存输出设置比特决定把何种输出信号进行锁存，我们可以对其进行编程。当锁存输出不同时，我们或者把输出信号送到输出缓冲器中，或者把触发器输出送到输出缓冲器。转换速率控制输出信号发生改变的速率。当输出驱动外部设备时，由于输出的快速变化可能会引起感应噪声，通过降低转换速率可以减少噪声。当上拉电阻启动时，一个上拉电阻就会被连接到 I/O 管脚上。此内部上拉电阻可以防止输入的漂移。图 3.39 中高亮线表示 I/O 模块中由三态使能端和正上拉电阻控制的输出路径。

I/O 标准

早期的 FPGA 是 TTL 和 CMOS 信号兼容的，现在 FPGA 的输入/输出信号的标准得到了进一步的增加。现代 FPGA 的 I/O 模块可以把信号转化为各种 I/O 信号标准，如

LVTTL:　低电平晶体管–晶体管逻辑

PCI： 周边元件扩展接口
LVCMOS： 低电压互补金属氧化物半导体
LVPECL： 低电平正发射极耦合逻辑
SSTL： 线脚系列终端逻辑
AGP： 加速图形接口
CTT： 中央中继终端
GTL： 射电收发逻辑电路
HSTL： 高速收发逻辑

这些标准中的一些使用 5 V，而一些使用 3.3 V 或甚至 1.5 V。LVTTL 就是一个能承受 5 V 信号的 3.3 V 标准的例子。LVCMOS2 是能承受 5 V 信号的 2.5 V 信号标准。PCI 标准有 5 V 和 3.3 V 两个版本。一些标准需要一个输入电压参考。

3.4.6 FPGA 中的专用元件

早期，FPGA 只是具有中低复杂度，集成了可编程 I/O 和接口的逻辑模块。现在，FPGA 生产厂商开始把嵌入式、DSP、专用乘法器、专用存储器、A/D 转换器等专用器件引入到 FPGA 中。这些专用器件的使用增加了 FPGA 的通用功能，例如，如果不使用专用乘法器，那么我们就得用通用逻辑模块来实现乘法器，这会降低设计的效率。

专用存储器

FPGA 的一个主要特点就是在芯片中内嵌了专用存储模块（RAM）。内嵌 RAM 用以实现电路所需的存储器。它可以视为一个表格，此表格可以在处理过程存储常数或系数。当我们用 FPGA 设计嵌入式时，它也可以用来实现存储器。现代 FPGA 内部通常含有 16K ~ 10M 的存储器。内嵌 RAM 的宽度必须要适应芯片的大小，通常可以作为 32K×1，16K×2，8K×4，4K×8 的存储器使用。FPGA 内部含有多个存储器模块，按不同方法放置可以得到各种比例的存储器。与各种比例存储器相对应的地址总线和数据总线个数如表 3.10 所示。

表 3.10 各种比例 RAM

宽　度	深　度	地址总线	数据总线
1	32K	15 位	1 位
2	16K	14 位	2 位
4	8K	13 位	4 位
8	4K	12 位	8 位
16	2K	11 位	16 位

专用算术单元

很多人用 FPGA 实现算术逻辑。如果直接在 FPGA 的逻辑模块中实现算术逻辑，那么将会占用大量的面积，消耗很大的功率，并且速度较慢。因此，当大多数用户需要使用算术模块（如加法器、乘法器等）时，如果芯片本身就含有此类专用模块，那么将会非常方便。大多数 FPGA 都内嵌快速进位逻辑，这样就可以构建快速加法器。现在的很多 FPGA 内部还含有专用乘法器（如表 3.11 所示）。这样，我们就可以不用把多个逻辑模块映射成乘法器，而直接可以使用 FPGA 内嵌的乘法器了。同时，使用内嵌的乘法器要比可编程逻辑构建的乘法器更加有效率。Xilinx 和 Altera 公司生产的 FPGA 中都提供了 18×18 乘法器（如表 3.11 所示）。

表 3.11 带有专用乘法器的 FPGA

FPGA	专用乘法器
Xilinx Virtex-4,	18×18 个乘法器
Virtex - II Pro/X,	
Spartan-3 E,	
Spartan 3/3L	
Altera	18×18 个乘法器
Stratix II	
Cyclone II	

DSP 模块

乘法操作是 DSP 中的基本操作，因此 FPGA 中的专用乘法器使 FPGA 可以作为 DSP 来使用。FPGA 生产厂商不仅内嵌乘法器，而且可以内嵌 DSP 模块，如 FFT 硬件模块、FIR 滤波器、IIR 滤波器等。FPGA 还可以提供加密和解密、压缩和解压缩、保密等功能。当 FPGA 内嵌大量的专用元件时，为了保证专用元件的使用，所以 FPGA 中很大的一部分都不被使用。例如，某些 FPGA 中，其对 DSP 的支持受专用乘法器的限制。

嵌入式

许多现代 FPGA 都包含一个完整的处理单元（如表 3.12 所示）。这一特点对于使用多器件的设计者来说是非常有用的。例如，我们要设计一个系统，系统的一部分要用硬件实现，另一部分要用可编程处理器实现。我们可以用微处理器实现需要很大灵活性的那部分电路，对于另外一部分电路，如果我们想获得更好的性能（对比于可编程处理器），那么可以用 FPGA 的逻辑模块来实现。有些 FPGA 还含有小型 MIPS 处理器核（如 MIPS R 4000），有些内嵌 IBM PowerPC 处理器，有些 FPGA 还含有自定义处理器，如 Xilinx 公司的 MicroBlaze 和 Altera 公司的 Nios 处理器。

表 3.12 带有内嵌处理器的 FPGA

FPGA	内嵌处理器
Xilinx	IBM 400 MHz
Virtex-4,	PowerPC
Virtex- II Pro/X	
Xilinx	MicroBlaze
Spartan-3 E,	PicoBlaze
Spartan 3/3I	
Altera	Nios II
Stratix II	
Cyclone II	
Altera	ARM,
APEX	MIPS,
APEX II	Nios
Altera	ARM 9
Excalibur	
Actel	ARM 7
Fusion	

内容寻址存储器

一些 FPGA 中的存储模块可以用做内容寻址存储器（CAM）。通常意义上的存储器是指用户提供一个地址，则得到与之相对应的存储单元和存储的内容。CAM 是一种特殊的存储器，它不是按地址检索的，而是按内容检索的。当我们提供一个数据单元时，CAM 可以给出此数据所在存储单元的地址。与 RAM 相比，CAM 包含更多逻辑。这是因为要对所有存储单元同时进行搜索以确定数据所在的地址。有些 FPGA 允许内嵌 CAM（如 Altera 公司的 APEX II）。

Actel 公司生产的基于熔丝结构的 FPGA

如图 3.40 所示的 FPGA 中提供了多种专用元件，如内嵌 RAM、解密、A/D 转换器。芯片的核心是逻辑模块片（Actel 公司的 VersaTiles 技术）。内嵌 RAM 在 SRAM 之上，逻辑模块片之下。多个专用元件分布在 SRAM 下面，接近芯片底部。此 FPGA 中包含一个使用 AES 算法进行解密的模块（AES 是高级加密标准的缩写，它由美国在 2001 年提出）。FPGA 中还有一个 A/D 转换器，它可以对模拟块接收到的电压、电流和温度等模拟信号进行处理。

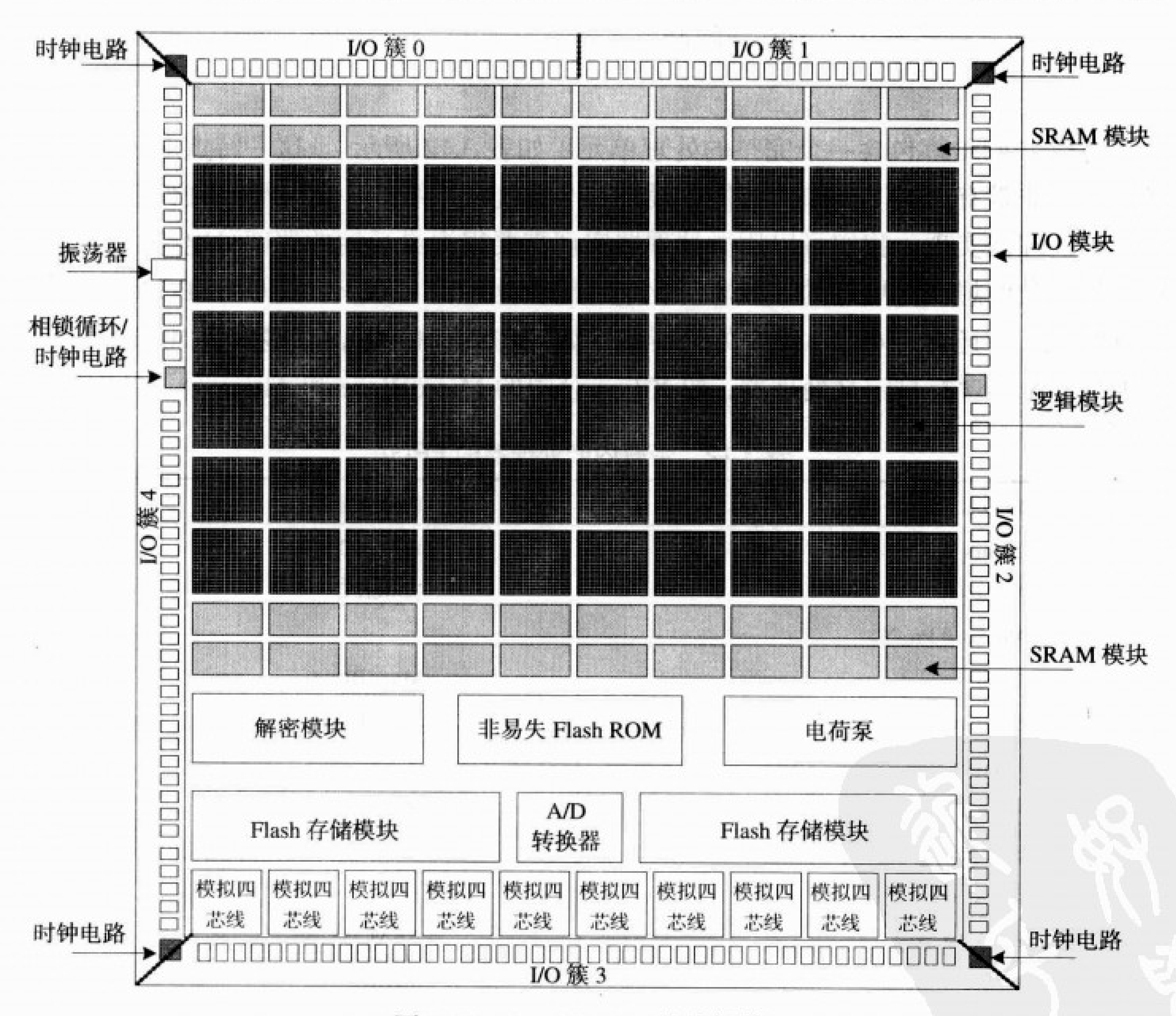


图 3.40　Actel Fusion 芯片概貌

3.4.7 FPGA 的应用

FPGA 的应用很广泛，它常用来实现各种电路。下面介绍它的几个主要应用。

快速制板

FPGA 常用来对大型系统进行快速制板。通过使用 FPGA，设计者可以很快地制造出满足要求的系统。由于 FPGA 包含 500 万或更多的门电路，所以仅使用一片 FPGA 就能构建很多大型实际系统。如果一片 FPGA 不够用，那么还可以把几片 FPGA 连接起来实现一个大型系统。通过把几个 FPGA 板放到一个底板上（母板），我们就可以完成大型系统的快速制板。

中速系统

FPGA 实现的电路基本上都在 150 ~ 200 MHz 的时钟频率范围内工作。如果我们要设计的系统工作在此速度下，那么无需制板，编程后的 FPGA 就可以作为最终的产品，此时我们只需更新软件就可以完成对系统的升级，无需进行硬件上的改变。现代 FPGA 的速度可以满足各种应用要求。

交互逻辑

FPGA 可以用来实现模块和元件之间的交互逻辑。如果交互协议或格式进行了微小的改变，则有必要设计新的交互逻辑。如果使用 SRAM FPGA 设计交互逻辑，那么只需更新软件就可以在同一片 FPGA 上设计出新的交互逻辑。

硬件加速器（协处理器）

如果在系统硬件上添加一个加速内核，那么此系统上应用软件的速度将得到提升。一片基于 SRAM、具有可重新设置功能的 FPGA 就可以很好地实现此加速内核。由于此内核是加速正在运行的应用软件，所以需要对 FPGA 进行实时动态更新，所以此类 FPGA 可以满足要求。基于 FPGA 的硬件可以应用于各个方面，如计算机结构仿真加速、嵌入式、硬件测试等。

3.4.8 FPGA 设计流程

精通 CAD 制图对可编程门阵列设计具有一定的帮助。我们可以用不同的方法进行设计。

在 FPGA 使用早期，设计从结构捕捉开始，甚至使用更低层面的工具。那时我们使用的工具是逻辑表达式、卡诺图等 FPGA 中的专用逻辑模块。结构捕捉是指构建出所设计系统的结构图，结构图上的标准硬件元件被输入到 CAD 软件上进行实现。

现在，我们可以使用自动合成工具，让 VHDL 硬件描述语言作为输入，在门电路与触发器直接构建接口以实现系统。行为描述方式 VHDL 语言可以很快地被翻译，并用于系统实现。在过去的十年中，合成工具得到了极大的发展。

使用 FPGA 设计数字系统遵循以下步骤：

1. 用行为描述、寄存器传输层面（RTL）或结构描述的硬件描述语言进行系统设计。
2. 设计仿真和调试。
3. 在目标器件上对设计进行合成。
4. 映射。此步骤把逻辑图分解成适合设置逻辑模块的小部分。
5. 布局和布线。此步骤将使逻辑模块处于 FPGA 的适当位置，并且对逻辑模块间的互联设置布线。
6. 生成可以编辑 FPGA 的位模型。
7. 将模型下载到 FPGA 的内部结构记忆单元并测试 FPGA 的操作。

在现代 CAD 工具中通常集成了步骤 3, 4, 5。但是，步骤中提到的各个过程是在一个步骤或是几个步骤中实现的，这与通用编译器集成了编译和汇编两个步骤是相同的。早期我们使用高级语言编译器时，编译这个词仅仅意味着把高级语言转化为汇编语言格式。把汇编语言转化为机器语言是由汇编器来完成的。现在，大多数高级语言的编译环境都集成了这两个步骤。

在基于 SRAM 的 FPGA 中，当最终的系统被建立后，编辑 FPGA 的数据模型被存储在 EPROM 中，并且当电源接通时会自动下载到 FPGA 中。EPROM 被连接到 FPGA 中，如图 3.41 所示。电源接通后，FPGA 就会复位。这时，它通过向 EPROM 输入中提供一系列地址从 EPROM 中读取数据，并且把 EPROM 的输出数据存储在 FPGA 内部存储器单元中。在基于 Flash 的 FPGA 中无需上述步骤，因为 Flash 是非易变器件。在反熔丝 FPGA 中，通过改变开关来配置比特。

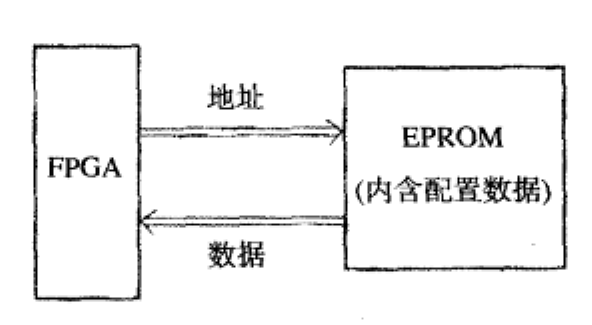


图 3.41 基于 SRAM 的 FPGA 初始化时 EPROM 的连接

本章中，我们介绍了多个不同类型可编程逻辑器件以及其在电路设计中的应用。首先我们介绍了早期的可编程逻辑器件，如 ROM，PAL 和 PLA；接着介绍了简单的 PLD 和 GAL。对于简单逻辑表达式的实现，在本章中也进行了讨论。然后，介绍了 CPLD 和 FPGA 的相关内容。介绍 FPGA 时，只讨论了 FPGA 的通用技术、不同结构和基本编程技术等。对于 FPGA 的具体内容，将在第 6 章中加以介绍。

习题

3.1 实现下列器件需要的最小 ROM 为多少？

(a) 一个 8 位全加器（具有前项进位和后项进位）

(b) 一个 BCD–二进制码转换器（2 个 BCD 码字）

(c) 一个 4 选 1 MUX

(d) 一个 32 位加法器（求两个 32 位数的和，结果为 33 位）

(e) 一个 3 线–8 线译码器

(f) 一个 32 位加法器（无前项进位和后项进位）

(g) 一个 16×16 位乘法器

(h) 一个 16 位全加器（具有前项进位和后项进位）

(i) 一个 8 线–3 线优先编码器

(j) 一个 10 线–4 线优先编码器

(k) 一个 8 线–1 线多路选择器

3.2 已知 $F = A'B'+BC'$ 且 $G = AC+B'$，写出用一个 8 字节×2 比特的 ROM 实现 F 和 G 的 VHDL 程序。程序中要包含 ROM 内容数组类型声明和常数声明。

3.3 使用一个 ROM 和两个 D 触发器实现下面的状态表，使用直接二进制状态赋值。

(a) 画出系统框图和 ROM 的真值表，真值表考察的各列分别为 $Q_1Q_0\ X\ D_1D_0Z$（并按此顺序进行考察）。

(b) 写出实现此系统的 VHDL 代码，使用一个数组表示 ROM 值，并使用两个进程。

当前状态	下一状态		当前输出(Z)	
	X = 0	X = 1	X = 0	X = 1
S_0	S_0	S_1	0	1
S_1	S_2	S_3	1	0
S_2	S_1	S_3	1	0
S_3	S_3	S_2	0	1

3.4 一个由 1 个 ROM 和 2 个 D 触发器（时钟上升沿触发）构成的电路的状态表如下所示：

Q_1Q_2	$Q_1^+Q_2^+$		Z	
	X = 0	X = 1	X = 0	X = 1
00	01	10	0	1
01	10	00	1	1
10	00	01	1	0

(a) 画出时序图。

(b) 写出此电路的 VHDL 程序。假设 ROM 的延迟为 10 ns，每个触发器的传输延迟为 15 ns。

3.5 写出实现下列方程的最小行 PLA 表：

$$f(A,B,C,D)=\sum m(3,6,7,11,15)$$
$$g(A,B,C,D)=\sum m(1,3,4,7,9,13)$$
$$h(A,B,C,D)=\sum m(4,6,8,10,11,12,14,15)$$

(a) 使用卡诺图找到相同项，并给出相同项逻辑表达式（其中相同项要求下划线），PLA 表以及 PLA 图（与图 3.15 类似）。

(b) 使用 *Logic Aid* 中的 Espresso 多输出化简路径，找到相同项，并把得到的结果与(a)中的结果比较，由于 *Logic Aid* Espresso 只找出最小行表，所以二者的结果可能不同，而且不必把每个 AND 项中的变量均最小化。

注意：在 *Logic Aid* 中变量必须命名为 *A, B, C, D, E, F, G* 和 *H*，不可以命名为 *X*1, *X*2, *X*3, *X*4。

3.6 写出实现下列方程的最小行 PLA 表：

(a) $f_1(A,B,C,D)=\sum m(0,2,3,6,7,8,9,11,13)$

$f_2(A,B,C,D)=\sum m(3,7,8,9,13)$

$f_3(A,B,C,D)=\sum m(0,2,4,6,8,12,13)$

(b) $f_1(A,B,C,D)=cd+ad+a'bc'd'$

$f_2(A,B,C,D)=bc'd'+ac'+ad'$

3.7 (a) 写出实现下列方程的最小行 PLA 表：

$$x(A,B,C,D)=\sum m(0,1,4,5,6,7,8,9,11,12,14,15)$$
$$y(A,B,C,D)=\sum m(0,1,4,5,8,10,11,12,14,15)$$
$$z(A,B,C,D)=\sum m(0,1,3,4,5,7,9,11,15)$$

(b) 如果要实现（a）中的各个等式，则 PLA 需如何链接？（与图 3.15 类似）

3.8 写出描述 22V10 输出宏单元（图 3.20 中用框标出）的 VHDL 程序，实体应包含 S_1 和 S_0，注意触发器具有异步复位（*AR*）和同步前置（*SP*）。

3.9 一个 *N* 位双向移位寄存器包含 *N* 位并行数据输入、*N* 位输出、1 个左串入输入（*LSI*）、1 个右串入输入（*RSI*）、1 个时钟输入和如下控制信号：

Load: 把并行数据输入载入移位寄存器中。

Rsh: 右移（*LSI* 进入最左一位）。

Lsh: 左移（*RSI* 进入最右一位）。

(a) 如果用 1 个 22V10 实现此寄存器，求 *N* 的最大值。

(b) 写出最右边两位的逻辑表达式。

3.10 用 CPLD 实现图 2.43 中的左移寄存器，并画出类似于图 3.25 的实现框图。给出触发器输入 D 的表达式。

3.11 一个用 22V10 实现的具有 4 个输出变量的 Mealy 时序电路，求其最多允许输入变量个数和最大状态数。其他任何具有相同输入和输出的 Mealy 电路是否均能由一个 22V10 实现？为什么？

3.12 (a) 传统门阵列和 FPGA 有什么不同？

(b) 按照结构不同进行分类，FPGA 可以分成几种类型？

(c) FPGA 有哪些不同的编程技术？

(d) SRAM FPGA 的主要优点是什么？

(e) 反熔丝 FPGA 的主要优点是什么？

(f) FPGA 中主要的可编程元件是什么？

(g) SRAM FPGA 的主要缺点是什么？

(h) 反熔丝 FPGA 的主要缺点是什么？

(i) 通常一个 SRAM 单元需要多少个晶体管构成？

(j) 什么是 MPGA？

(k) CPLD 和 FPGA 有什么不同？

(l) 对比于 FPGA，CPLD 有什么优点？

(m) 对比于 CPLD，FPGA 有什么优点？

(n) 写出三个 CPLD 生产商的名字。

(o) 写出三个 FPGA 生产商的名字。

3.13 (a) 在何种应用情况下，设计者应使用 CPLD，而不是 FPGA？

(b) 在何种应用情况下，设计者应使用 MPGA，而不是 FPGA？

(c) 在何种应用情况下，设计者应使用 FPGA，而不是 MPGA？

(d) 一个公司正在设计一款试验产品，此款产品要进行反复修改。如果用 FPGA 实现此产品，则应采用何种 FPGA（SRAM 还是反熔丝）？

(e) 一个公司正在设计一款试验产品，此款产品实际采用 FPGA 实现，并通过反复修改验证，证明其非常稳定。如果芯片面积和成品的最小化对公司很重要，则应选用何种 FPGA（SRAM 或 antifuse）？

(f) 一个公司正在设计一款试验产品，而且预计可以卖出 1000 个此种产品，此公司应选用 MPGA 还是 FPGA 来实现此产品？

(g) 一个公司正在设计一款试验产品，而且预计可以卖出 100 个此种产品，此公司应选用 MPGA 还是 FPGA 来实现此产品？

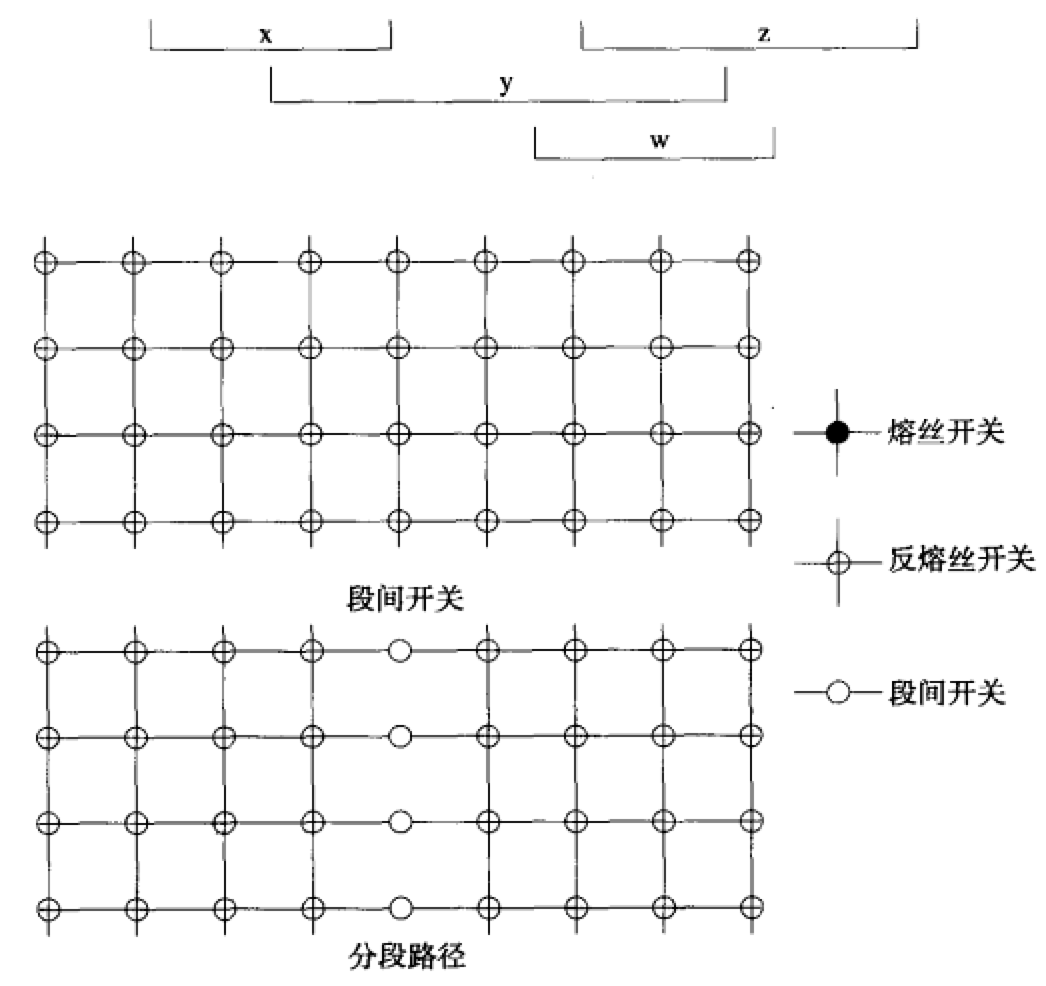
3.14 (a) 用 FPGA 实现方程 $F_1 = A'BC+B'C+AB$，此 FPGA 要求具有由 4 选 1MUX 构成的可编程逻辑模块。

(b) 用一个 MUX 实现方程 $F1 = A'B+AB'+AC'+A'C$，需使用 MUX 的最小面积为多少？假设

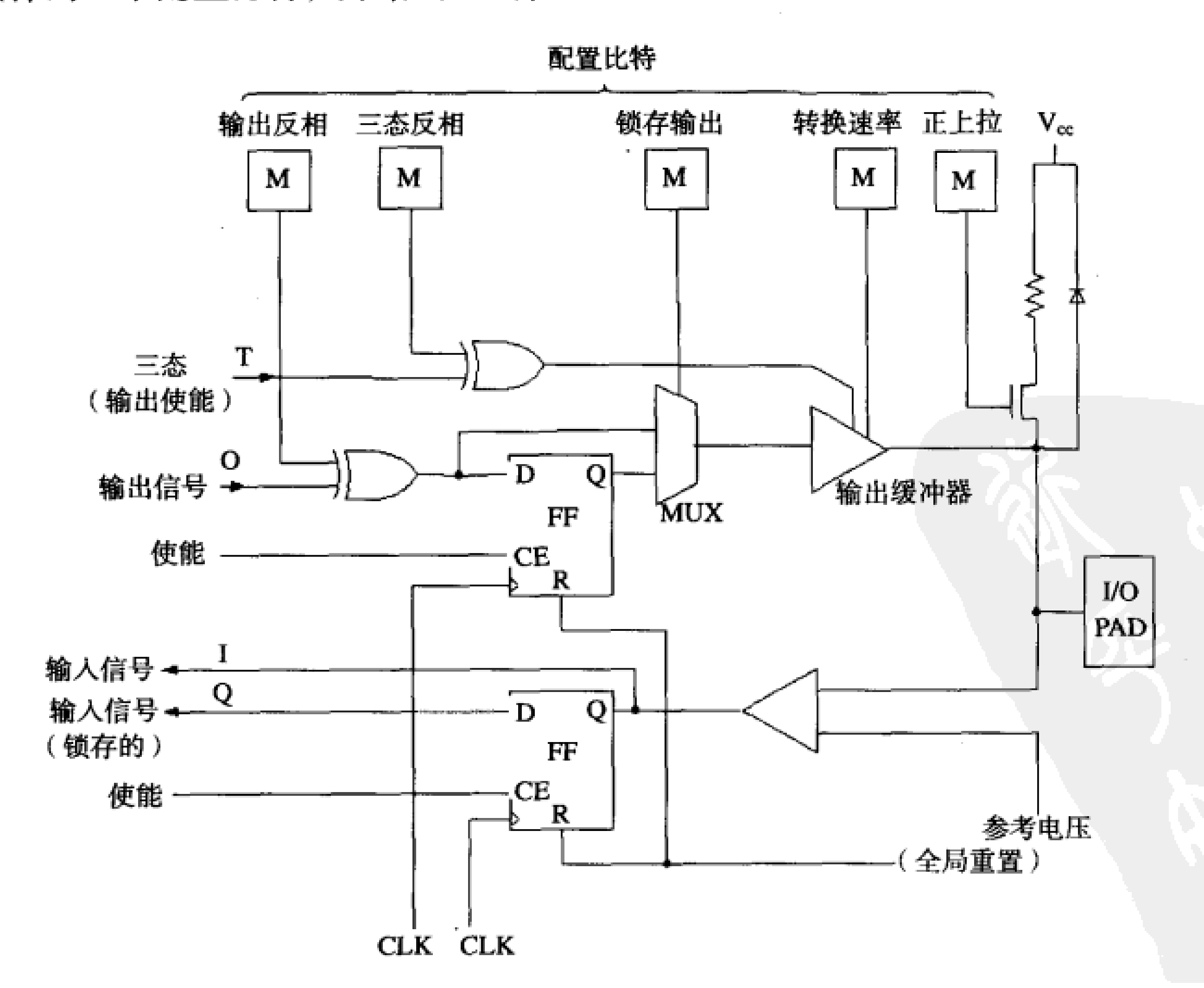
输入与其补值均可用。

3.15 (a) 在下面的非分段路径上实现‘w’, ‘x’, ‘y’和‘z’网的布线。要求使用尽可能少的路径。

(b) 在下面的分段路径上实现‘w’, ‘x’, ‘y’和‘z’网的布线。要求使用尽可能少的路径。



3.16 观察下面的可编程 I/O 模块，要求把此 I/O 模块配置成输入管脚，请用高亮线标出链接，给出具体的 5 个配置比特，并给出 T 的值。



第4章 设 计 举 例

为了说明如何设计小型数字系统，本章中我们列举了多个 VHDL 设计实例。我们给出的设计思路是把一个设计分为两部分：一部分是对控制器的设计，另一部分是对数据通道的设计。控制器用于控制数字系统的操作顺序。我们使用行为描述方式的 VHDL 代码对数字系统进行描述，这样我们就可以对系统进行仿真，并对所用的算法进行验证。同时，我们也介绍如何使用结构描述方式，对于具有特定硬件结构的系统进行编程实现。

在任何设计中，你首先充分理解问题和弄清设计规范。如果问题还没有说清楚，那么你应该努力弄清设计的特点。在实际设计中，如果其他项目组或是客户公司委派你们进行设计，那么你一定要弄清设计规范，这样会减少后续设计过程中很多麻烦。好的设计是从清晰的规范文档开始的。

一旦问题叙述明确后，设计者通常开始考虑完成指定设计所需的基本模块。设计者通常会考虑标准组成模块，如加法器、移位寄存器、计数器等。在传统设计方法中，把所设计的系统分为“数据通道”和“控制器”两个部分。数据通道是指实际进行数据处理的硬件部分，控制器则给数据通道发送控制信号或者命令，并从数据通道以状态信号的形式得到反馈，如图 4.1 所示。

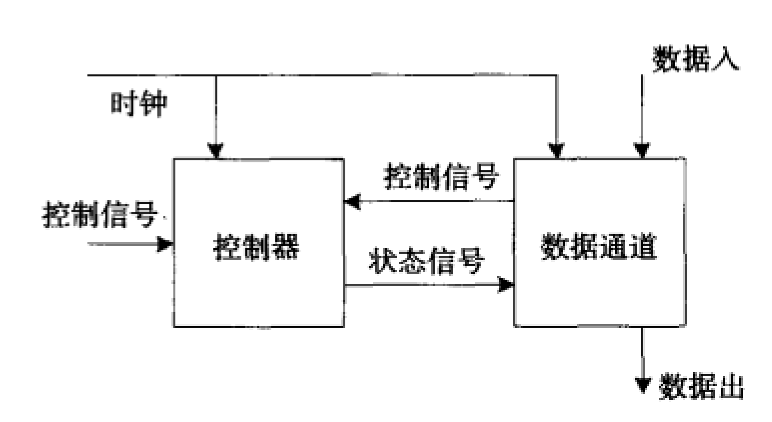


图 4.1 把一个设计分为数据通道和控制器

在微处理器中，数据通道是算术逻辑单元（ALU），它完成所有关键操作处理。控制器是控制逻辑单元，给数据通道发送适当的控制信号，指示数据通道进行加法、乘法、移位或者指令要求的其他操作。不少人把数据通道与数据总线混为一谈，但是在传统设计术语中，数据通道是指实际的数据处理单元。

弄清数据通道和控制器之间的区别对调试很有帮助（比如找出设计中的错误），而且对设计的修改也有益。在许多情况下，我们修改设计只需修改控制单元，因为同一个数据通道可以满足新的要求。这样控制器产生一系列新的控制信号以完成修改的新功能。一个设计往往需要对数据通道和控制器进行反复修改优化。

本章中，我们将讨论各种设计实例，既有算术实例，也有非算数实例。非算数实例包括七段译码器设计、交通灯设计、记分板设计和键盘扫描器设计。算术实例包括加法器设计、乘法器和除法器设计。

4.1 BCD 码-七段显示译码器

七段数码管显示器通常用于显示数字计数器、数字手表和数字钟上的显示数字。通过七段数码管的不同组合，一个数字手表可以显示时间。本例子中，每段数码管的标记和每个数字的表示如图 4.2 所示。

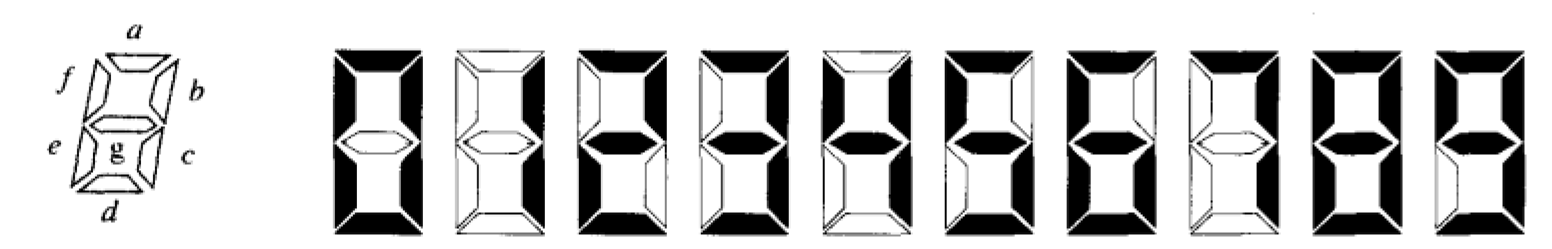


图 4.2　七段数码管显示

下面我们设计一个 BCD 码–七段显示译码器。BCD 是指用二进制编码的十进制数。在该格式下，每个十进制数都用 4 位二进制数来表示。该译码器为组合逻辑电路，因此不必使用状态机。译码器模块如图 4.3 所示，译码器每次只能显示一个 BCD 数字。

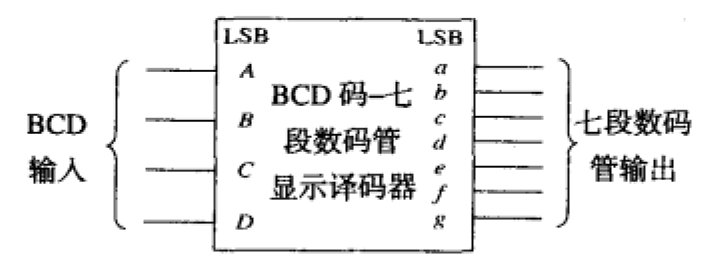


图 4.3　BCD 码–七段显示译码器

我们用行为描述方式编写该译码器的 VHDL 结构体，只用了一个进程，并用 CASE 语句描述了这一组合电路，如图 4.4 所示。图中进程语句的敏感信号表为输入的 BCD 数（4 位）。

```
entity bcd_seven is
  port(bcd: in bit_vector(3 downto 0);
      seven: out bit_vector(7 downto 1));
                -- LSB is segment a of the display. MSB is segment g
end bcd_seven;

architecture behavioral of bcd_seven is
begin
  process(bcd)
  begin
    case bcd is
      when "0000" => seven <= "0111111";
      when "0001" => seven <= "0000110";
      when "0010" => seven <= "1011011";
      when "0011" => seven <= "1001111";
      when "0100" => seven <= "1100110";
      when "0101" => seven <= "1101101";
      when "0110" => seven <= "1111101";
      when "0111" => seven <= "0000111";
      when "1000" => seven <= "1111111";
      when "1001" => seven <= "1101111";
      when others => null;
    end case;
  end process;
end behavioral;
```

图 4.4　BCD 码-七段显示译码器的行为描述 VHDL 代码

4.2　BCD 加法器

本例中，我们要设计一个两个 BCD 数字加法器。该加法器可以把两个 BCD 数相加，并得到 BCD 码格式的和数。在 BCD 码中，每个十进制数都是用二进制编码的，例如，十进制数 97 用 BCD 码表示为 1001 0111，其中前 4 位表示数字 9，后 4 位表示数字 7。注意 BCD 码表示的 97 与二进制码表示的 97 不同。在二进制中十进制数 97 表示为 110001，而且只有 7 位。BCD 码不使用以下 4 位二进制组合：1010, 1011, 1100, 1101 和 1110（对应于十六进制中的 A ~ F）。由于 4 位二进制数可以表示的 16 个数中有 6 个没有使用，所以对于同一个数来说，其 BCD 码表示要比其二进制表示占用更多的位数。

当对 BCD 码进行加法运算时，对和数的每个数字都要进行调整以避开那 6 个未使用的码。例如，计算 6+8，结果是 14，其二进制表示为 1110，但用 BCD 码表示其和，则其最低 4 位应该表示数字 4，即为 0100。因此，为了得到正确的结果，只要当和数大于 9，我们就把和数再加上 6。图 4.5 给出了两位 BCD 数字求和的硬件结构。我们用二进制加法器把两个 BCD 数的低 4 位相加，如果其和超过 9，则再加上 6，并生成一个进位用于下一个高 4 位数字相加。BCD 高位数字的加法与此类同。

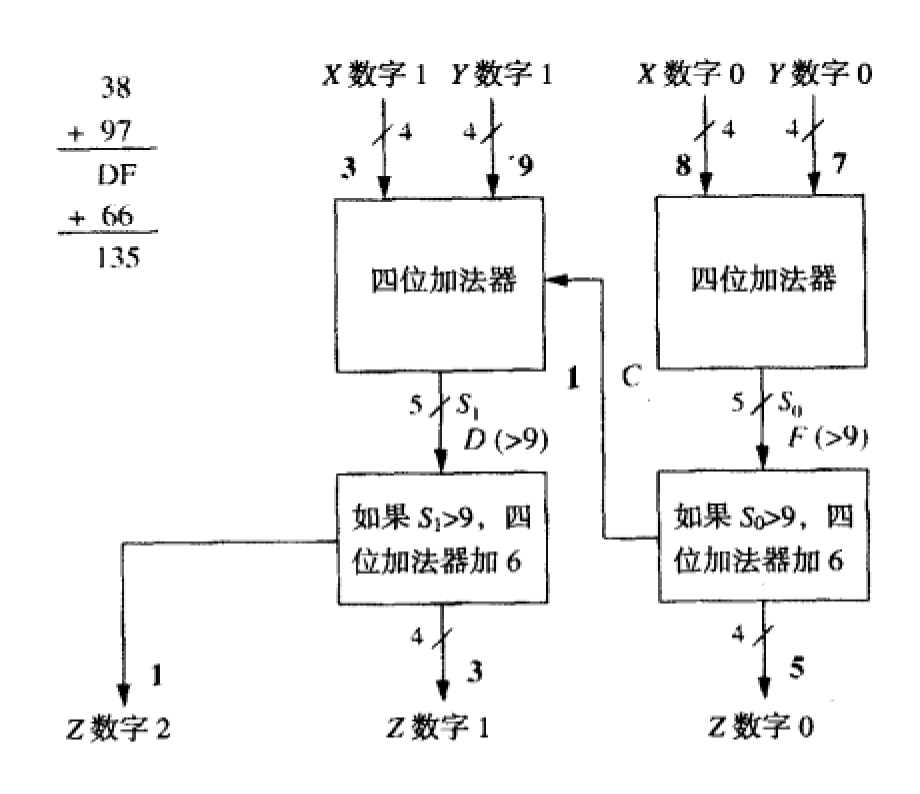


图 4.5　两个 BCD 数相加

BCD 码加法器的 VHDL 代码示于图 4.6。输入为 BCD 数字，用 *X* 和 *Y* 表示。两个两位 BCD 数相加，得到的结果可能超过两位，所以为和数预留 3 个 BCD 数字位，用 *Z* 表示，并且 *X*, *Y* 和 *Z* 的数据类型均为 IEEE numeric_bit library 中的无符号数。我们使用 alias 语句定义每个 BCD 数的每个数字，例如，用 *Xdig1* 表示输入 *X* 的高位数字，其 VHDL 语句为

```
alias Xdig1: unsigned(3 downto 0) is X(7 downto 4);
```

这条语句表示无论何时，只要我们提到 *Xdig1*，它都表示输入 *X* 的高位数字。如果计算 BCD 数 97 和 38 的和，则和数为 135，因此 *Zdig2* = 1，*Zdig1* = 3，*Zdig0* = 5。

```
library IEEE;
use IEEE.numeric_bit.all;

entity BCD_Adder is
  port(X, Y: in unsigned(7 downto 0);
       Z: out unsigned(11 downto 0));
end BCD_Adder;

architecture BCDadd of BCD_Adder is
alias Xdig1: unsigned(3 downto 0) is X(7 downto 4);
alias Xdig0: unsigned(3 downto 0) is X(3 downto 0);
alias Ydig1: unsigned(3 downto 0) is Y(7 downto 4);
alias Ydig0: unsigned(3 downto 0) is Y(3 downto 0);
alias Zdig2: unsigned(3 downto 0) is Z(11 downto 8);
alias Zdig1: unsigned(3 downto 0) is Z(7 downto 4);
alias Zdig0: unsigned(3 downto 0) is Z(3 downto 0);
signal S0, S1: unsigned(4 downto 0);
signal C: bit;
begin
  S0 <= '0'&Xdig0 + Ydig0; -- overloaded +
  Zdig0 <= S0(3 downto 0) + 6 when S0 > 9
      else S0(3 downto 0); -- add 6 if needed
  C <= '1' when S0 > 9 else '0';
  S1 <= '0' & Xdig1 + Ydig1 + unsigned'(0 =>C);
                           -- type conversion done on C before adding
  Zdig1 <= S1(3 downto 0) + 6 when S1 > 9
      else S1(3 downto 0);
  Zdig2 <= "0001" when S1 > 9 else "0000";
end BCDadd;
```

图 4.6　BCD 加法器的 VHDL 代码

IEEE numeric_bit 库中‘+’运算符可以用于对每个 BCD 数字进行加运算。两个 4 位向量相加，其和为 5 位向量，且和暂时存储在 *S0* 和 *S1* 中（*S0* 和 *S1* 均定义为 5 位向量）。由于我们要求结果为 5 位，所以必须在 *Xdig0* 上拼接一个 ‘0’（*Ydig0* 将自动匹配扩展）。因此语句

```
S0<= '0'&Xdig0 + Ydig0;
```

实现了最低有效数字相加。在进行第二个数字相加时，不仅要把这两个数字相加，而且要加上 *Xdig0* 和 *Ydig0* 相加后得到的进位。进位 *C* 在加到 *Xdig1* + *Ydig1* 之前一定要转换为无符号数字类型。Unsigned'(0=>C)可以实现数字类型的转换。这样，第二个数字相加可以用如下语句实现：

```
S1 <='0'&Xdig1 + Ydig1 + unsigned'(0=>C);
```

4.3　32 位加法器

下面我们设计一个 32 位加法器。实现该加法器的最简单的方法就是构建一个行波进位加法器（ripple-carry adder），如图 4.7 所示。这种加法器就是把 32 个 1 位全加器连接起来构成一个 32

位加法器，进位从最低有效位开始“波动”传递到最高有效位。

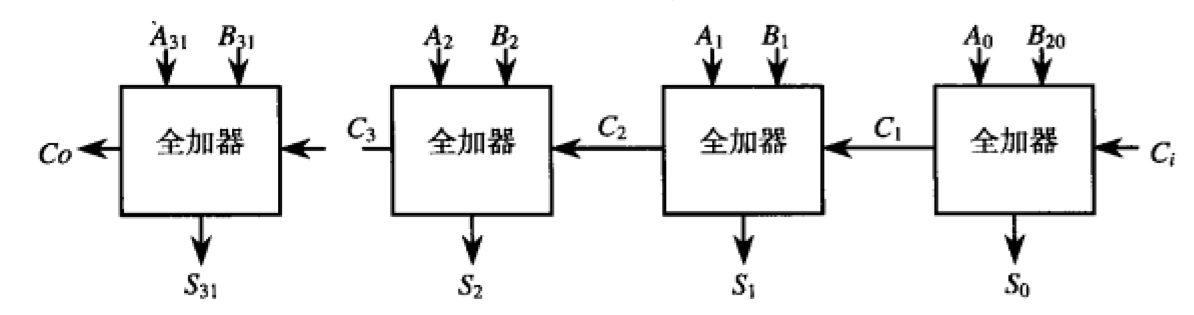


图 4.7　32 位行波进位加法器

如果门延迟为 t_g，则 1 位加法器的延迟为 $2t_g$（假设和数与进位都是用与或表达式表示的，而且忽略反相器的门延迟），那么一个 32 位行波进位加法器大概有接近 64 个门延迟。因此，如果一个门延迟为 1 ns，则 32 位行波进位加法器的最大工作频率约为 16 MHz。对于很多应用来说，这一频率是远远不够的。因此，设计者通常求助于更快的加法器。

4.3.1　先行进位加法器

先行进位（carry look-ahead adder, CLA）加法是一种很流行的快速计算加法的技术。在先行进位加法器中，根据输入信号提前计算进位信号。对于任意数字位 i，当对应的输入（例如 A_i 和 B_i）均为‘1’时，就会产生进位；或者当输入中有一个为‘1’，且有前一级的进位时，也会产生进位。也就是说，当 A_i 和 B_i 都为‘1’时，不管有否前一级的进位，都产生进位；当 C_i =1，且 A_i 和 B_i 中有一个为‘1’时，也产生进位。因此，第 i 级产生的进位表达式为

$$C_{i+1} = A_iB_i + (A_i \oplus B_i) \cdot C_i \tag{4.1}$$

“⊕”表示异或操作。式 4.1 表示了第 i 级可以自身生成进位（A_i= ‘1’），或者是由较低位的进位“传递”到第 i 级（$(A_i \oplus B_i) \cdot C_i$）而生成进位。

由于 A_iB_i = ‘1’时就产生进位，所以我们可以得到一般**生成函数**（G_i）如下：

$$G_i = A_iB_i \tag{4.2}$$

同理，由于 $(A_i \oplus B_i)$ 表示是否应该把低级进位传递下去，因此我们可以得到一般**传递函数**（P_i）如下：

$$P_i = A_i \oplus B_i \tag{4.3}$$

我们可以看出传递函数和生成函数都只与输入比特有关，而且其实现只有一个或两个门延迟。由于当 A_i 和 B_i 都为‘1’，或 A_i 和 B_i 中有一个为‘1’时就会生成进位，所以我们也可以把传递函数写为

$$P_i = A_i + B_i \tag{4.4}$$

其中，用或操作取代了异或操作。逻辑上，该函数仍可以正确产生进位，但是按习惯我们还是用异或来定义传递函数，以表示单纯的进位传递（而不包含它自己产生进位的情况）。一般和信号也可以表示为

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i \tag{4.5}$$

把式(4.2)和式(4.3)代入式(4.1)中，我们可以把进位表达式重写为

$$C_{i+1} = G_i + P_iC_i \tag{4.6}$$

在 4 位加法器中，我们可以通过反复使用式(4.6)得到每个进位 C_i 的表达式：

$$C_1 = G_0 + P_0C_0 \tag{4.7}$$

$$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0 \tag{4.8}$$

$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \tag{4.9}$$

$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_0 + P_3P_2P_1P_0C_0 \tag{4.10}$$

这些进位就是用 P_i, G_i 和 C_0 表示的先行进位。如此一来，我们可以直接计算任意级的和与进位，而无需等待进位从前面的各个级逐级传递过来。由于 P_i 和 G_i 只会引起 1 ~ 2 个门延迟，所以在 3 ~ 4 个门延迟后我们就可以得到进位 C_i。与行波加法器相比，本算法的优点在于其每级进位所需延迟是相同的，且这些延迟与相加数据的位数无关。当然，我们需要另外使用一些门计算先行进位。现在，我们可以构建一个 4 位先行进位加法器了，如图 4.8 所示。

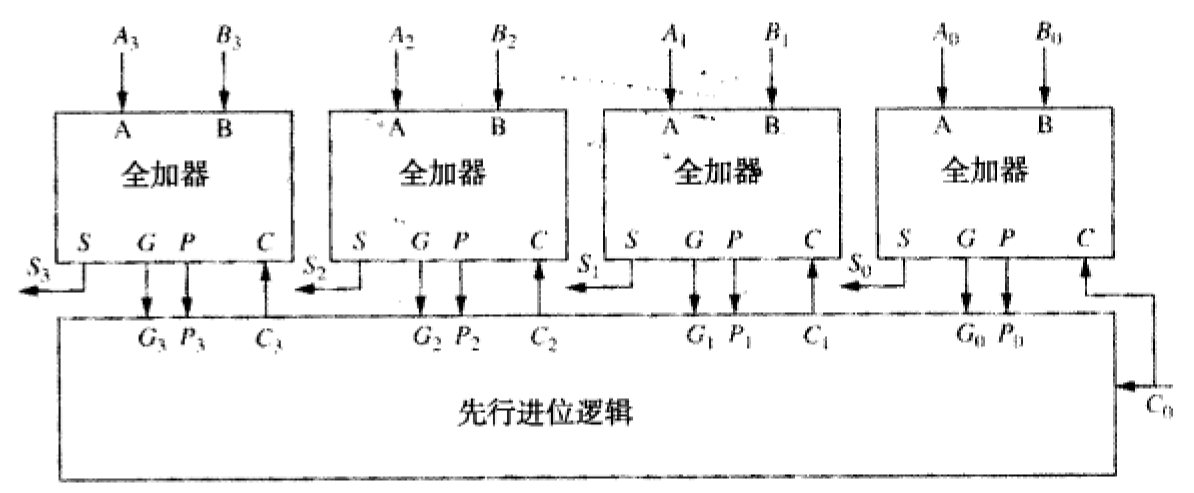


图 4.8 4 位 CLA 实现框图

先行进位加法器的缺点在于先行进位逻辑，如式(4.7) ~ 式(4.10)所示并不简单，当位数超过 4 位时，会变得很复杂。因此，先行进位加法器的实现通常以 4 位加法器为基本模块，以分层结构实现位数为 4 的倍数的加法器。图 4.9 为一个 16 位先行进位加法器。与图 4.8 相似，图 4.9 也使用了 4 个先行进位加法器。这里我们没有每 1 位都计算一次先行进位，而是每 4 位计算一次，并通过组传递信号 P_G 和组生成信号 G_G 计算得到下一级先行进位逻辑所需的先行进位。若每个组的内部传递信号是正确的，则整个组的传递信号也是正确的；在一组中，如果最高有效位（MSB）生成一个进位，则此组的生成信号是正确的，或者当一个较低位生成一个进位并且所有高于它的位都传递了此进位时，此组的生成信号也是正确的。由此可得

$$P_G = P_3P_2P_1P_0 \tag{4.11}$$

$$G_G = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 \tag{4.12}$$

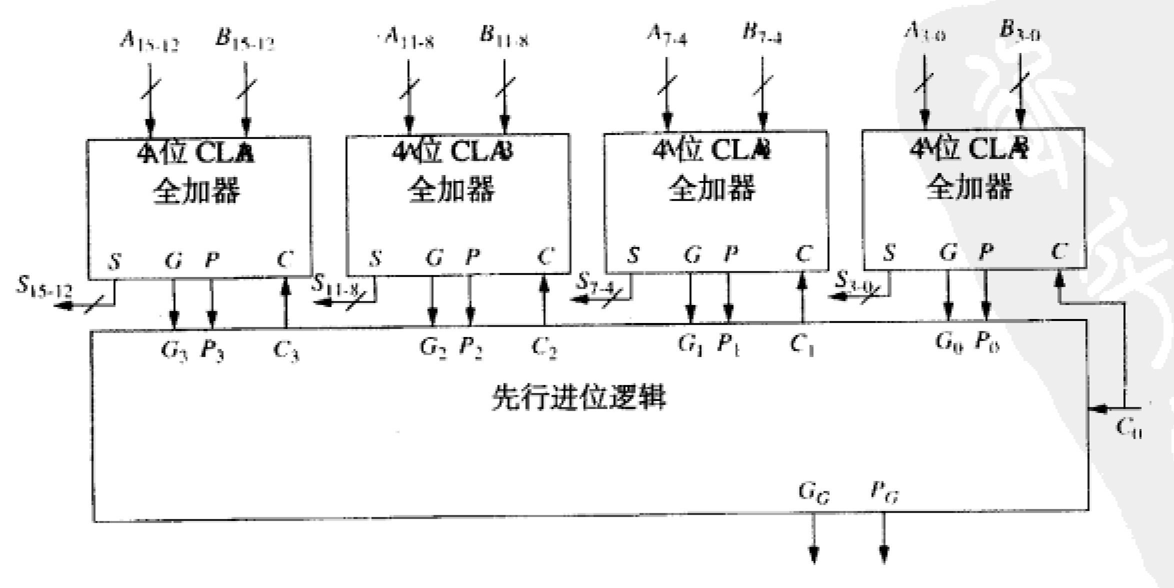


图 4.9 16 位 CLA 实现框图

在 3 ~ 4 个门延迟后，我们就可以得到组传递信号 P_G 和组生成信号 G_G（比得到 P_i 和 G_i 信号多 1 ~ 2 个门延迟）。4 位先行进位加法器的 VHDL 描述如图 4.10 所示。

```
entity CLA4 is
  port(A, B: in bit_vector(3 downto 0); Ci: in bit;          -- Inputs
       S: out bit_vector(3 downto 0); Co, PG, GG: out bit);  -- Outputs
end CLA4;

architecture Structure of CLA4 is
component GPFullAdder
  port(X, Y, Cin: in bit;     -- Inputs
       G, P, Sum: out bit);  -- Outputs
end component;
component CLALogic is
  port(G, P: in bit_vector(3 downto 0); Ci: in bit;          -- Inputs
       C: out bit_vector(3 downto 1); Co, PG, GG: out bit);  -- Outputs
end component;

signal G, P: bit_vector(3 downto 0); -- carry internal signals
signal C: bit_vector(3 downto 1);
begin      --instantiate four copies of the GPFullAdder
  CarryLogic: CLALogic port map (G, P, Ci, C, Co, PG, GG);
  FA0: GPFullAdder port map (A(0), B(0), Ci, G(0), P(0), S(0));
  FA1: GPFullAdder port map (A(1), B(1), C(1), G(1), P(1), S(1));
  FA2: GPFullAdder port map (A(2), B(2), C(2), G(2), P(2), S(2));
  FA3: GPFullAdder port map (A(3), B(3), C(3), G(3), P(3), S(3));
end Structure;

entity CLALogic is
  port(G, P: in bit_vector(3 downto 0); Ci: in bit;          -- Inputs
      C: out bit_vector(3 downto 1); Co, PG, GG: out bit);  -- Outputs
end CLALogic;

architecture Equations of CLALogic is
signal GG_int, PG_int: bit;
begin      -- concurrent assignment statements
  C(1) <= G(0) or (P(0) and Ci);
  C(2) <= G(1) or (P(1) and G(0)) or (P(1) and P(0) and Ci);
  C(3) <= G(2) or (P(2) and G(1)) or (P(2) and P(1) and G(0)) or
          (P(2) and P(1) and P(0) and Ci);
  PG_int <= P(3) and P(2) and P(1) and P(0);
  GG_int <= G(3) or (P(3) and G(2)) or (P(3) and P(2) and G(1)) or
          (P(3) and P(2) and P(1) and G(0));
  Co <= GG_int or (PG_int and Ci);
  PG <= PG_int;
  GG <= GG_int;
```

图 4.10　4 位先行进位加法器的 VHDL 代码

```
end Equations;

entity GPFullAdder is
  port(X, Y, Cin: in bit;    -- Inputs
       G, P, Sum: out bit);  -- Outputs
end GPFullAdder;

architecture Equations of GPFullAdder is
signal P_int: bit;
begin                    -- concurrent assignment statements
  G <= X and Y;
  P <= P_int;
  P_int <= X xor Y;
  Sum <= P_int xor Cin;
end Equations;
```

图 4.10（续） 4位先行进位加法器的VHDL代码

16位先行进位加法器可以由4个4位先行进位加法器和1个先行进位逻辑模块组成，在此基础上再加一层先行进位逻辑模块就可以得到64位先行进位加法器。当加法器的位数从16变为64时，只增加了2个门延迟。作为练习，大家可以自己编写一下16位先行进位加法器的VHDL代码。

图4.11是32位加法器的行为描述方式VHDL代码，其中使用了IEEE numeric_bit库中的'+'重载操作符。当此代码进行综合时，设计者使用的工具和技术决定此加法器是行波进位加法器还是快速双层加法器。不同的拓扑结构有不同的面积、功率和延迟特性。

```
library IEEE;
use IEEE.numeric_bit.all;

entity Adder32 is
  port(A, B: in unsigned(31 downto 0); Ci: in bit;  -- Inputs
       S: out unsigned(31 downto 0); Co: out bit);  -- Outputs
end Adder32;

architecture overload of Adder32 is
signal Sum33: unsigned(32 downto 0);
begin
  Sum33 <= '0' & A + B + unsigned'(0 =>Ci);        -- adder
  S <= Sum33(31 downto 0);
  Co <= Sum33(32);
end overload;
```

图 4.11 32位加法器行为描述模块

例 4.1 若门延迟为t_g，则最快的32位加法器的延迟为多少？假设不考虑硬件消耗，只考虑速度。

解：我们可以用输入位的与或式表示32位加法器的每个和位，一共可以写出33个这样的表

达式，其中有 1 个是进位输出表达式。这些表达式都很长，有些表达式的乘积项甚至有 60 多个变量。若门电路的输入可以任意多，则理论上可以构建一个双层结构加法器。虽然此加法器可能无法实现，但是理论上来说，如果门延迟为 t_g，则该快速加法器的延迟为 $2t_g$。

例 4.2 行波进位加法器是最小的 32 位加法器吗?

解：一个 32 位行波进位加法器需要使用 32 个 1 位加法器。我们也可以用一个 1 位全加器实现一个 32 位串行加法器。输入数据被一位一位地传入全加器中，每次只传 1 位。每两位相加产生的进位输出存储在触发器中，并反馈回全加器作为下一次加法运算的进位输入。硬件实现见图 4.12。加法器的延迟为 $32(2t_g+t_{ff})$，其中 $2t_g$ 是 1 位加法器的延迟，t_{ff} 是触发器的延迟（包括启动时间在内）。如果触发器的延迟最少为 2 个门延迟，那么此 32 位串行加法器的延迟最少为 $128t_g$。此加法器的硬件结构很简单，但是还需要一个控制器生成 32 个移位信息，而且存储操作数的寄存器也必须具备移位存储能力。

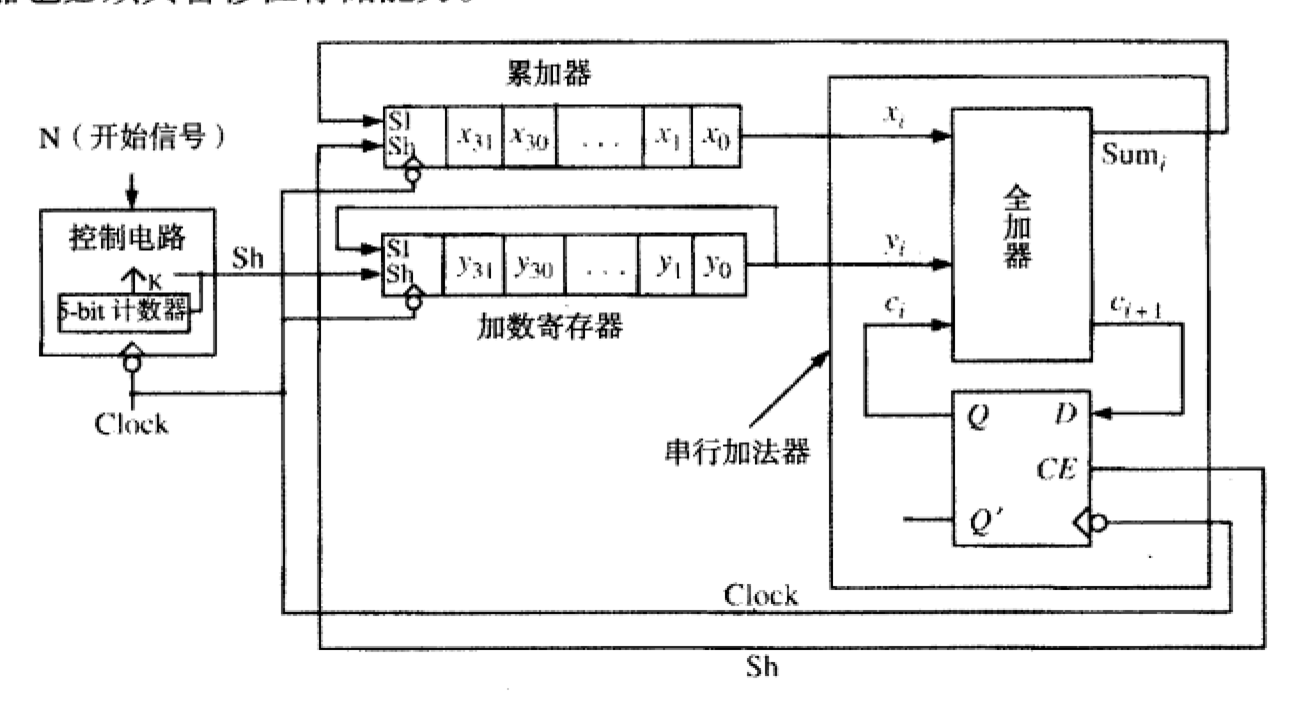


图 4.12 用一个 1 位加法器构建一个 32 位串行加法器

由于我们之前讨论的延迟特性的存在，因此，即使按数据流描述方式编写的 VHDL 程序（见图 4.10）进行综合时，也不能保证综合器生成一个先行进位加法器。对于具有某些特殊硬件元件的目标技术来说，我们也许可以通过软件对综合器的输出进行优化。例如，如果选用支持快速加法器的 FPGA，软件可以把一部分功能映射到快速加法电路中实现。由于使用的 FPGA 逻辑模块和互连的数目不同，所以延迟的长度可能与手工计算得到的结果不同。表 4.1 给出了基于门电路实现的、不同大小的行波进位加法器、先行进位加法器和串行加法器的延迟。从表中我们可以看出当构建大型加法器时，先行进位加法器的延迟性能是有吸引力的。

表 4.1 行波加法器和先行进位加法器的比较

加法器大小	行波进位加法器延迟	CLA 延迟	串行加法器延迟
4 位	$8\ t_g$	$5\text{-}6\ t_g$	$16\ t_g$
16 位	$32\ t_g$	$7\text{–}8\ t_g$	$64\ t_g$
32 位	$64\ t_g$	$9\text{–}10\ t_g$	$128\ t_g$
64 位	$128\ t_g$	$9\text{–}10\ t_g$	$256\ t_g$

4.4　交通灯控制器

下面我们为 A, B 两路的交叉路口设计一个时序交通灯控制器。每一条马路均有交通传感器，它能检测是否有车辆靠近或停在交叉路口。S_a=1 表示 A 路有车靠近，S_b=1 表示 B 路有车靠近。A 路是主干道，若 B 路无车靠近，则 A 路的绿灯一直亮。若 B 路有车靠近，则 A 路交通灯改变，B 路绿灯亮。50 秒后，如果 B 路上还有车，而 A 路上无车，则 B 路循环再延长 10 秒；其他情况时，交通灯变回。A 路的通车时间最短为 60 秒。超过 60 秒后，若 B 路有车，则 A 路交通灯改变。图 4.13 给出了控制器的外部连接。三个输出（G_a, Y_a 和 R_a）分别驱动 A 路的绿灯、黄灯和红灯。另外三个输出（G_b, Y_b 和 R_b）驱动 B 路中相应的灯。

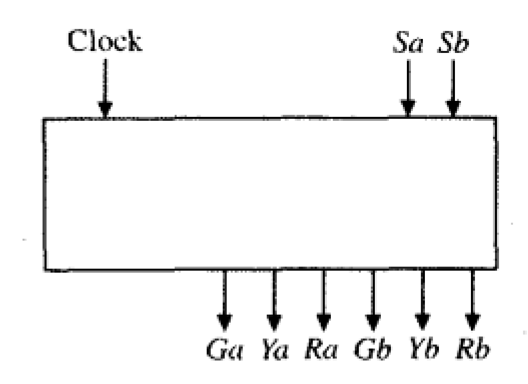


图 4.13　交通灯控制器的框图

图 4.14 给出了控制器的 Moore 状态图。为了时序控制，时序电路由一个周期为 10 秒的时钟驱动。这样，至多每隔 10 秒钟，状态就会变化一次。图中的符号 G_aR_b 表示 G_a= R_b =1，而其他输出变量均为 0。弧上的 $S_a'S_b$ 表示 S_a=0 且 S_b=1 时，将沿弧进行状态转移。弧上若无标记，则表示在有效时钟到来时就发生状态转移，与输入变量取值无关。这样，A 道绿灯将保持 6 个时钟周期（60 秒），然后在 B 道有车时变成黄灯。

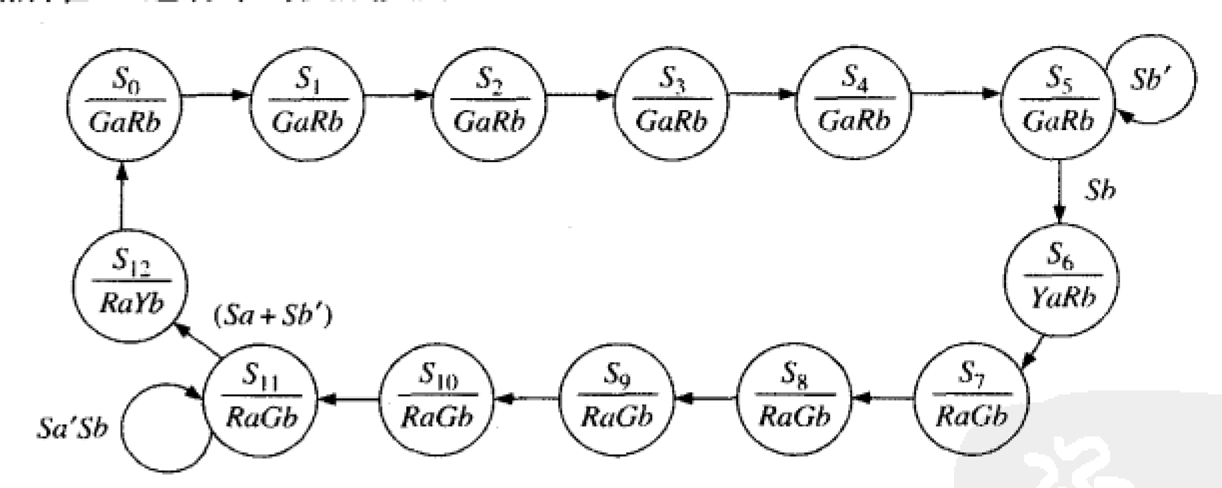


图 4.14　交通灯控制器的状态图

交通灯控制器的 VHDL 代码（图 4.15）使用了两个进程来描述此状态机。无论何时，只要状态 S_a 或 S_b 发生变化，则第一个进程将更新输出和下一状态。当时钟上升沿到达时，第二个进程将更新状态寄存器。Case 语句中使用了带范围的 when 语句。由于从状态 S_0 到状态 S_4 的输出相同，并且每一个状态的下一状态都是顺序排列的，所以我们使用了一条带范围的 when 语句，而不是 5 条单独的 when 语句：

```
when 0 to 4 => Ga<= '1'; Rb<= '1'; nextstate<= state + 1;
```

对于每一个状态，在 case 语句中只列了取值为‘1’的信号。由于在 VHDL 语言中，在信号

发生改变之前将保持原值，所以转到下一状态时，我们应当关闭每个信号。例如，在状态 6，我们应当置 G_a 为 ‘0’；在状态 7，我们应当置 Y_a 为 ‘0’；等等。我们可以通过在 when 语句中插入适当的语句来实现这一目的，例如我们可以在 when 6=>语句中插入 G_a<= ‘0’。一个关闭输出信号的更简便的方法是在 case 语句前将它们全部置 ‘0’，如图 4.15 所示。开始时，由于我们把本来应为 ‘1’ 的信号置成 ‘0’，所以可能会出现毛刺。然而，由于进程中的顺序语句都是瞬时执行的，所以它并不会造成问题。例如，假设在 20 ns 时刻，状态从 S_2 变到 S_3。原本 G_a 和 R_b 均为 ‘1’，但当进程开始执行，第一行代码被执行时，G_a 和 R_b 将在 20+Δ时刻变为 ‘0’。然后 Case 语句开始执行，G_a 和 R_b 应该在 20+Δ时刻变为 ‘1’。由于这两个时刻是同一时间，所以新值（ ‘1’ ）覆盖了原先的值（ ‘0’ ），信号也不会再变为 ‘0’ 了。

```
entity traffic_light is
  port(clk, Sa, Sb: in bit;
       Ra, Rb, Ga, Gb, Ya, Yb: inout bit);
end traffic_light;

architecture behave of traffic_light is
signal state, nextstate: integer range 0 to 12;
type light is (R, Y, G);
signal lightA, lightB: light;  -- define signals for waveform output
begin
  process(state, Sa, Sb)
  begin
    Ra <= '0'; Rb <= '0'; Ga <= '0'; Gb <= '0'; Ya <= '0'; Yb <= '0';
    case state is
      when 0 to 4 => Ga => '1'; Rb => '1'; nextstate => state+1;
      when 5 => Ga <= '1'; Rb <= '1';
        if Sb = '1' then nextstate <= 6; end if;
      when 6 => Ya <= '1'; Rb <= '1'; nextstate <= 7;
      when 7 to 10 => Ra <= '1'; Gb <= '1'; nextstate <= state+1;
      when 11 => Ra <= '1'; Gb <= '1';
        if (Sa='1' or Sb='0') then nextstate <= 12; end if;
      when 12 => Ra <= '1'; Yb <= '1'; nextstate <= 0;
    end case;
  end process;
  process(clk)
  begin
    if clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;
  lightA <= R when Ra='1' else Y when Ya='1' else G when Ga='1';
  lightB <= R when Rb='1' else Y when Yb='1' else G when Gb='1';
end behave;
```

图 4.15　交通灯控制器的 VHDL 代码

在完成交通控制器的设计之前，我们将检验其 VHDL 代码，看它是否满足要求。最低要求，我们使用的检测序列应当可以使状态图中的每个弧至少都遍历一次。我们还应再进行一些测试以检验多种交通情况时的时序，例如 A 道和 B 道都交通拥挤时、两道车辆都很少时、仅 A 道交通拥挤时、仅 B 道交通拥挤时，以及一些特殊情况，如一辆车在绿灯时出现故障、一辆汽车在红灯时闯过交叉路口等。

为了更方便的表示仿真器的输出，我们定义了一个名为 light 的数据类型和两个信号 *light*A 和 *light*B。light 数据类型有 R, Y, G 三个值，可以为信号 *light*A 和 *light*B 赋值。在 A 灯为红灯时，我们编写代码把 *light*A 置为 R，当灯为黄时置为 Y，当灯为绿时置为 G。下面的仿真器命令文件首

先检测图表中两个自循环均被遍历的情况，然后检测两个自循环均未被遍历的情况。

```
add wave clk SA SB state lightA lightB
force clk 0 0, 1 5 sec -repeat 10 sec
force SA 1 0, 0 40, 1 170, 0 230, 1 250 sec
force SB 0 0, 1 70, 0 100, 1 120, 0 150, 1 210, 0 250, 1 270 sec
```

图 4.16 的检验结果证明，交通灯在特定时刻能够发生变化。

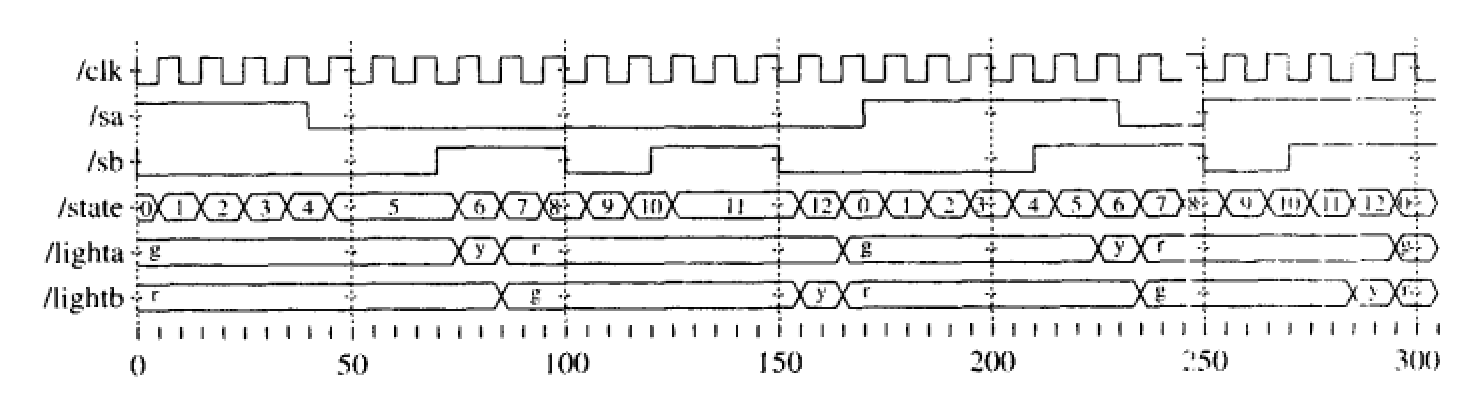


图 4.16 交通灯控制器的检验结果

4.5 控制电路状态图

在介绍下一个例子之前，我们首先介绍一下控制状态图中使用的符号，随后指出构建一个正确的状态图所必须满足的条件。通常，我们使用变量名，而不是‘0’和‘1’来标注控制状态图。这样就使状态图更易读些，特别是当输入和输出数目较多时。如果在状态图中，我们在弧线上标注 X_iX_j/Z_pZ_q，这表示如果 X_iX_j 为 1（不管其他的输入值如何），那么输出 Z_p 和 Z_q 为 1（其他输出为 0），并沿着弧线到达下一个状态。例如，一个电路有四个输入(X_1, X_2, X_3, X_4)，四个输出(Z_1, Z_2, Z_3, Z_4)，符号 X_1X_4'/Z_2Z_3 等价于 1--0/0110。通常来说，如果用输入表达式 I 标注弧线，当 I=1 时则沿着弧线进入下一个状态。例如，如果输入的标注为 $AB+C'$，那么当 $AB+C'$=1 时，我们将沿着这条弧线到达下一个状态。

为了得到完整、具体、正确的状态图，即图中对于每种输入组合其下一个状态都是唯一确定的，就必须对每个状态 S_k 的输入标注给予以下限制：

1. 如果 I_i, I_j 是当前状态 S_k 的任意一对输入标注，那么当 $i \neq j$ 时，I_iI_j=0。
2. 如果从 S_k 有 n 条弧线发出，这 n 条弧线对应的输入标注分别为 $I_1, I_2, \cdots, I_n$，则 $I_1+I_2+\cdots+I_n=1$。

条件 1 保证在任意给定时刻至多只有一个输入标注为 1，条件 2 保证在任意给定的时刻最少有一个输入标注为 1。因此仅有一个标注将是 1，且对于每种输入组合都会有唯一的下一状态。例如，考虑图 4.17 所示状态图的一部分，其中 $I_1=X_1, I_2=X_1'X_2', I_3=X_1'X_2$。

图 4.17 中，对于状态 S_k，条件 1 和条件 2 均满足。

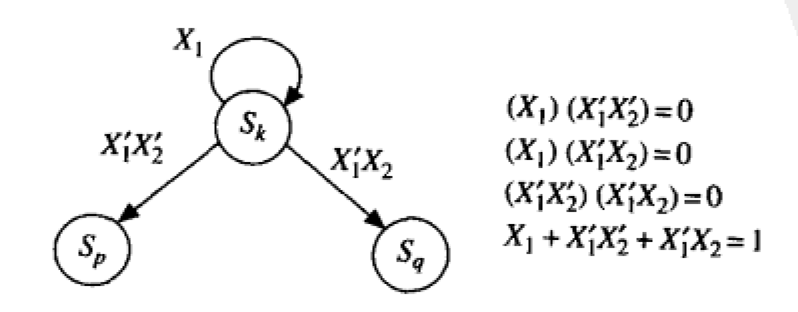


图 4.17 状态图的一部分

即使一个不完整的状态图，也必须总是满足条件 2 的，且对于状态 S_k，所有输入值的组合必须满足条件 1。因此，只有状态 S_k 的输入组合 $X_1=X_2=1$ 不发生，图 4.18 才会是正确的。

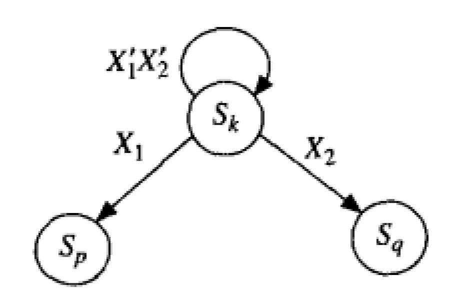


图 4.18 状态图的一部分

如果有三个输入变量(X_1, X_2, X_3)，则前面的部分状态图可以由下面的状态表表示：

	000	001	010	011	100	101	110	111
S_k	S_k	S_k	S_q	S_q	S_p	S_p	—	—

4.6 记分板和控制器

下面我们介绍一个可以显示 0 ~ 99（十进制数）的简单的记分板系统。此系统的输入必须含有一个复位信号和两个用于递增或递减记分的控制信号。当递增控制信号为真时，记分板上数字加 1；当递减控制信号为真时，记分板上数字减 1；如果递增和递减控制信号同时为真，则什么也不做。

当前分数用七段显示电路显示。如果要清除记分板上数字，则复位键必须被按下连续 5 个周期，这样可以防止由于不慎碰触而造成的数字擦除。计分板数字应该允许递减，以纠正误加分（针对不慎多加分的情况）。

4.6.1 数据通道

本设计的核心部分是一个两位 BCD 数字计数器用以计分，还需要两个七段显示器用以显示分数，也需要两个译码器把每个 BCD 码数字转换为七段数码显示。该系统的框图如图 4.19 所示。因为真正的复位，只有按下复位键连续 5 个时钟周期后才发生，所以我们还要使用一个 3 位复位计数器（称为 *rstcnt*）。

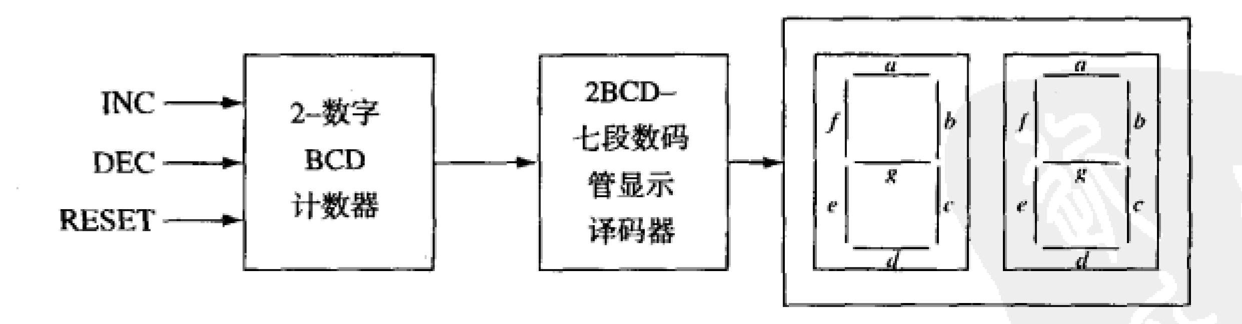


图 4.19 简单记分板模型

4.6.2 控制器

本电路的控制器工作如下。这一有限状态机（FSM）只有两个状态，如图 4.20 所示。初始状态（S_0）时，BCD 计数器清零，复位计数器也清零。随后，FSM 进入下一个状态（S_1）。此状态中，计数器开始计数。在每个时钟周期，根据输入信号计数器加 1 或减 1；如果有复位信号 *rst*，

则 *rstcnt* 加 1；若复位计数器已经为 4，且在第五个时钟周期时仍存在复位信号，则系统由状态 S_1 变为状态 S_0。当 *inc* 信号出现，而 *dec* 信号未出现时，BCD 计数器递增计数，图中右上角弧线处的 *add1* 表示 BCD 计数器递增计数；当 *dec* 信号出现，而 *inc* 信号未出现时，BCD 计数器递减计数，图中右下角弧线处的 *sub1* 表示 BCD 计数器递减计数。如果在任何时钟周期内，只要 *rst* 信号没有出现，则 *rstcnt* 计数器就清零。如果 *inc* 信号和 *dec* 信号同时出现或均不出现，则 *rstcnt* 计数器也清零，而 BCD 计数器则不变。

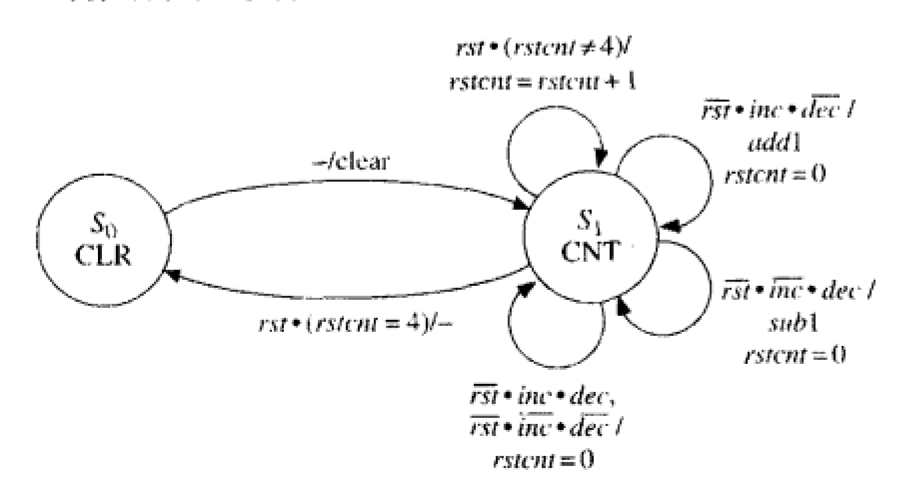


图 4.20　记分板系统的状态图

4.6.3　VHLD 模型

该记分板系统的 VHDL 代码见图 4.21。*seg7disp1* 和 *seg7disp2* 分别表示两个七段显示器，在程序中说明为无符号 7 位向量。七段显示器的七个数码管分别记为 *a* ~ *g*，如图 4.19 所示。由于使用了无符号数，所以我们可以使用'+'操作符使计数器加 1。七段显示译码器可以用数组或查找表实现。查找表中有 10 个 7 位向量（表示 0 ~ 9 个数）。我们定义了一个新的数据类型 *sevsegarray*，为 10 元素二维数组，且每个元素均为 7 位无符号向量。我们用 *sevsegarray* 存放与每个 BCD 数字相对应的七段数值。查找表的地址必须是用整数数据类型表示的，所以使用 to_integer 转换函数生成数据所在位置的数组的下标。语句 To_integer(BCD0)表示把 BCD0 转换为整数类型。下面的语句

```
seg7disp0  <=  seg7rom(to_integer(BCD0));
```

表示在数组 *seg7rom* 中读出相应的元素，并把十进制数字转换为 7 段形式。BCD 加操作用过载运算符'+'实现。如果当前计数器的值小于 9，则加 1；如果等于 9，则加 1 后，个位变为 0 而十位数加 1。减操作同理。

```
library IEEE;
use IEEE.numeric_bit.all; -- any package with overloaded add and subtract

entity Scoreboard is
  port(clk, rst, inc, dec: in bit;
       seg7disp1, seg7disp0: out unsigned(6 downto 0));
end Scoreboard;

architecture Behavioral of Scoreboard is
signal State: integer range 0 to 1;
signal BCD1, BCD0: unsigned(3 downto 0) : = "0000"; -- unsigned bit vector
```

图 4.21　记分板系统的 VHDL 程序

```
signal rstcnt: integer range 0 to 4 : = 0;
type sevsegarray is array (0 to 9) of unsigned(6 downto 0);
constant seg7Rom: sevsegarray : =
  ("0111111", "0000110", "1011011", "1001111", "1100110", "1101101", "1111100",
   "0000111", "1111111", "1100111"); -- active high with "gfedcba"order
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      case State is
        when 0 => -- initial state
          BCD1 <= "0000"; BCD0 <= "0000"; -- clear counter
          rstcnt <= 0; -- reset RESETCOUNT
          State <= 1;
        when 1 => -- state in which the scoreboard waits for inc and dec
          if rst = '1' then
            if rstcnt = 4 then -- checking whether 5th reset cycle
              State <= 0;
            else rstcnt <= rstcnt + 1;
            end if;
          elsif inc = '1' and dec = '0' then
            rstcnt <= 0;
            if BCD0 < "1001" then
              BCD0 <= BCD0 + 1; -- library with overloaded " + " required
            elsif BCD1 < "1001" then
              BCD1 <= BCD1 + 1;
              BCD0 <= "0000";
            end if;
          elsif dec = '1' and inc = '0' then
            rstcnt <= 0;
            if BCD0 > "0000" then
              BCD0 <= BCD0 - 1; -- library with overloaded "-" required
            elsif BCD1 > "0000" then
              BCD1 <= BCD1 - 1;
              BCD0 <= "1001";
            end if;
          elsif (inc = '1' and dec = '1') or (inc = '0' and dec = '0') then
            rstcnt <= 0;
          end if;
      end case;
    end if;
  end process;
  seg7disp0 <= seg7rom(to_integer(BCD0)); -- type conversion function from
  seg7disp1 <= seg7rom(to_integer(BCD1)); -- IEEE numeric_bit package used
end Behavioral;
```

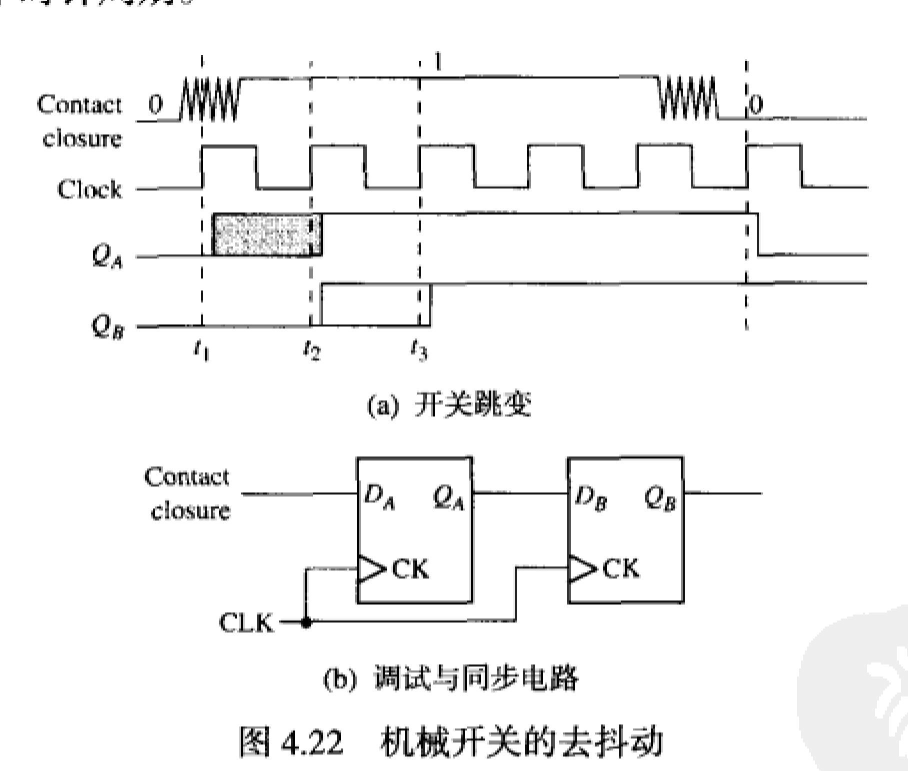
图 4.21（续） 记分板系统的 VHDL 程序

4.7　同步与去抖动

上节介绍的记分板系统中，*inc*, *dec* 和 *rst* 信号都是外部输入，而一个系统往往要涉及对外部输入的同步问题。键盘或按键开关并不一定与系统时钟信号同步，因此当它们作为同步时序电路的输入时，就必须对其进行同步。

涉及系统外部输入的另一个问题是开关抖动问题。当一个机械开关进行开闭操作时，开关触点会抖动，给开关输出引入噪声，如图 4.22(a)所示。这一开关触点达到最后稳定位置之前，触点抖动将持续几毫秒。当我们检测到一个开关闭合之后，最后读取键盘之前，必须等到这一抖动过程结束。任何含有机械开关的电路，都必须去掉开关的抖动。去抖动就是指去掉开关输出中的瞬态。

对触点进行同步和去抖动时，触发器是非常有用的器件。图 4.22(b)给出了一个去抖动和同步电路。在这种设计中，时钟周期要比抖动时间长。如果在抖动时，时钟上升沿到达，则触发器在 t_1 时刻锁存 0 或 1 值。如果 0 被锁存，那么在下一个时钟有效沿到来时（t_2），触发器锁存 1。因此，Q_A 可以看做是开关输出的同步和去抖动的信号。然而，如果开关改变时刻距时钟沿过近，那么就会不满足触发器的建立时间和保持时间，Q_A 也会有问题，可能会振荡或出现工作故障。尽管这种情况发生的可能性较低，但是为了保险起见，我们最好再加一个触发器。通过选择适当的时钟周期，我们使 Q_A 的振荡在下一个时钟有效沿到来之前完全消失。这样，输入 D_B 在时钟有效沿到来时一定是稳定的，去抖动的信号 Q_B 总是干净的，并与时钟信号总是同步的。但是 Q_B 要比开关按下的时刻延迟两个时钟周期。



(a) 开关跳变

(b) 调试与同步电路

图 4.22　机械开关的去抖动

4.7.1　单脉冲发生器

在记分板的设计中，我们假设每当存在 *inc* 和 *dec* 信号时，它们只持续一个时钟周期。数字系统的运行速度要比人工操作的速度快得多，所以人工提供一个只持续一个时钟脉冲的信号是很困难的。如果按键时间比一个时钟周期长，则在上例中，计数器将持续递增计数。如何解决这个问题呢？我们可以设计一个单脉冲发生器，当操作者按键或关闭开关时，该电路可以生成一个单脉冲信号。这种单脉冲发生器可以广泛用于各种人工按键或人工控制开关的场合。

现在，我们来设计一个单脉冲发生器电路。该电路可以在按键时刻传递一个与时钟周期等长

的同步脉冲。因此，电路必须能够感知键是否被按下并生成一个与时钟周期相同的输出信号。这一输出信号在按键松手之前不会再激活。

下面我们画出单脉冲发生器的状态图。单脉冲发生器必须有两个状态。一个状态检测是否有按键，另一个状态则检测按键是否结束。设第一个状态为 S_0，第二个状态为 S_1，SYNCPRESS 表示同步的按键操作。当电路处于状态 S_0，且键被按下时，系统生成一个单脉冲并进入状态 S_1。当系统从状态 S_0 变为 S_1 时，此单脉冲输出。只要系统处于状态 S_1 时，它就等待按键的结束。当按键结束时，系统从状态 S_1 变为起始状态 S_0，并等待再次按键。单脉冲的输出，只在从状态 S_0 变为 S_1 的时候才产生。系统的状态图见图 4.23。

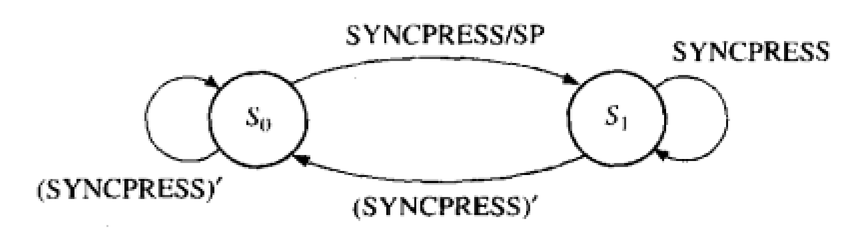


图 4.23　单脉冲发生器的状态图

由于此电路只有两个状态，所以在实现时可以只使用一个触发器。单脉冲发生器的实现框图见图 4.24。图中第一个模块是由图 4.22(b)中的电路实现的，它可以生成一个同步的按键信号 SYNCPRESS。触发器实现该状态机的两个状态。假设状态赋值为 S_0=0，S_1=1，此时触发器输出 Q 与 S_1 同步，触发器的反相输出 Q' 与 S_0 同步。由此，我们可以得到单脉冲 SP 表达式：

$$\text{SP} = \text{S0} \cdot \text{SYNCPRESS}$$

此外，我们应该注意到 $S_0 = S_1'$。三个触发器（图中的触发器和同步模块内部的两个触发器）可以提供一个去抖动的同步单脉冲信号。如果按键的输出经过这一电路，我们就可以得到去抖动的与时钟同步的一个单脉冲。如果使用这种电路连接外部按键信号是一个很好的工程实现，它可以保障操作的可控性和和可预测性。

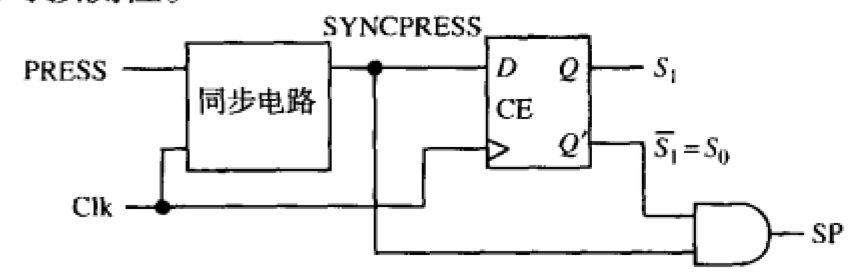
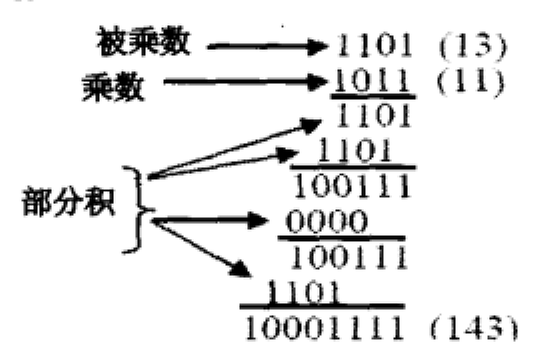


图 4.24　单脉冲发生器的和同步器电路

4.8　相加-移位结构乘法器

在这一节，我们将设计一个二进制无符号数乘法器。当我们做 A×B 时，第一个操作数(A)称为被乘数，第二个操作数(B)称为乘数。我们将看到，二进制乘法仅需要进行移位和加法运算。下面我们用二进制乘法计算 $13_{10} \times 11_{10}$：



注意到部分积是被乘数(1101)进行适当移位后的结果，或者是 0。每个部分积一形成立刻加到一起，而不是等得到所有的部分积后再加到一起，这样做避免了一次计算两个以上的二进制数加法。

两个 4 位数的乘法需要一个 4 位被乘数寄存器、一个 4 位乘数寄存器、一个 4 位全加器和一个用于存储乘积的 8 位寄存器。乘积寄存器作为一个累加器来累加部分积的和。如果在被乘数加到累加器前，被乘数每次都进行左移，那么就需要一个 8 位的加法器。因此最好每次对乘积寄存器右移，其实现框图见图 4.25 所示。这种类型的乘法器有时被称做串-并乘法器，因为乘数每位是串行处理的，而加法是并行操作的。图中箭头指出，累加器(ACC)的 4 位和被乘数寄存器的 4 位连接到加法器的输入；加法器和数的 4 位输出和 1 位进位再连回到累加器。当出现一个加信号(*Ad*)时，在下一个时钟脉冲，将加法器的输出转移到累加器中，这样把被乘数加到累加器中去。乘积寄存器的最左边多出来一位，用来暂存被乘数加到累加器时得到的进位。当出现一个移位信号(*Sh*)时，在下一个时钟脉冲，ACC 的全部 9 位右移 1 位。

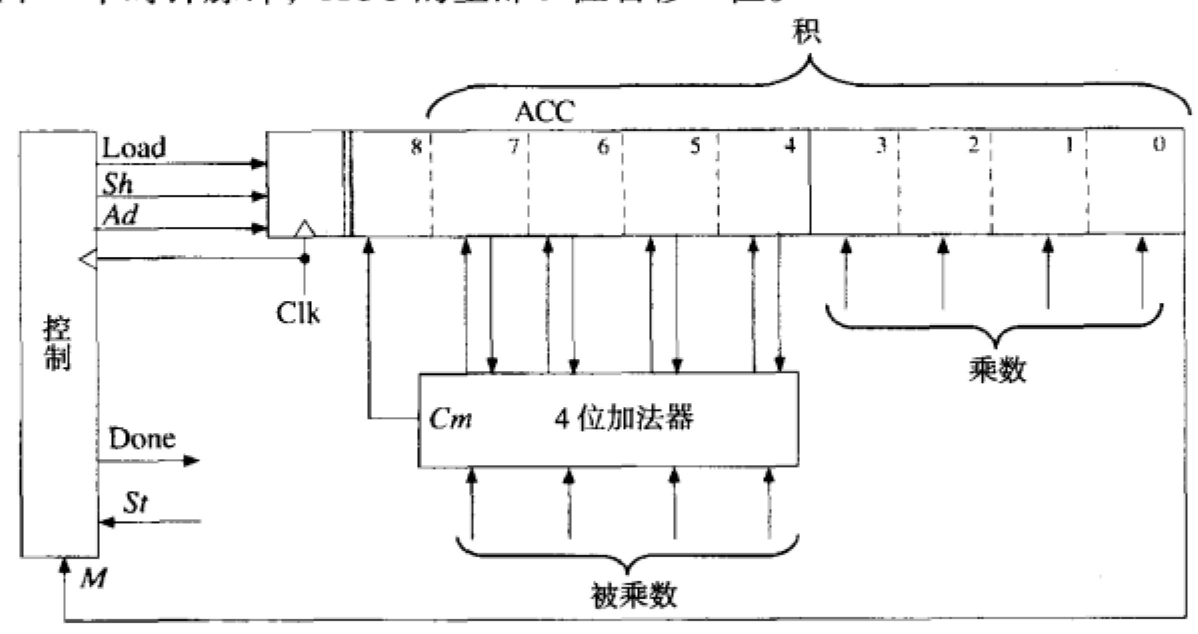
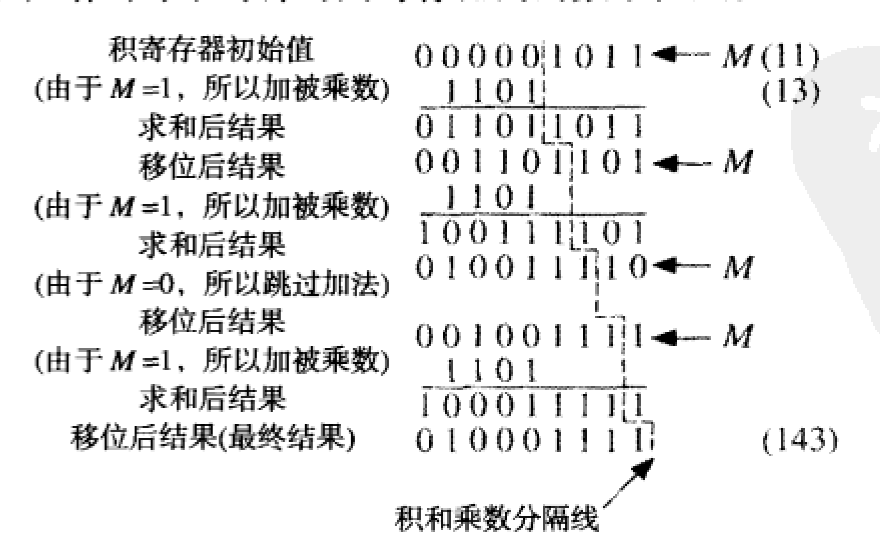


图 4.25 二进制乘法器框图

因为乘积寄存器的低四位开始时没有使用，所以将乘数存储在这个位置，这样就不用再使用一个单独的寄存器了。乘数的每一位用完之后，就可以移出寄存器的最右边的一位来保存增加的乘积位。在下一个时钟沿，移位信号(*Sh*)使乘积寄存器的内容（包括乘数）右移一位。控制电路在收到开始信号(*St*=1)后，就输出适当的相加和移位的控制信号序列。如果当前的乘数(*M*)为 1，则被乘数被加到累加器并右移一位；如果乘数为 0，则跳过加法操作，只进行右移。前面的 13×11 乘法例子重新操作如下，给出每个时钟时间寄存器内的数字位置。



控制电路必须设计成能输出正确的相加和移位的控制信号序列。图 4.26 是控制电路的状态图。在图 4.26 中，S_0是复位状态，收到一个开始信号(*St*=1)之前，电路一直处于 S_0状态。收到开

始信号后，电路就产生一个 *Load* 信号，使乘数读入累加器(ACC)的低 4 位中，而累加器的高 5 位清零。在状态 S_1，乘数(M)的最低位被测试。如果 M=1，则产生加信号；如果 M=0，则产生移位信号。类似的，在状态 S_3, S_5, S_7，通过测试当前乘数位 M 来决定生成加信号还是移位信号。在生成加信号后，总要产生移位信号（状态 S_2, S_4, S_6 和 S_8）。经过 4 次移位，控制电路进入状态 S_9，并回到状态 S_0 之前，生成 *Done* 信号。

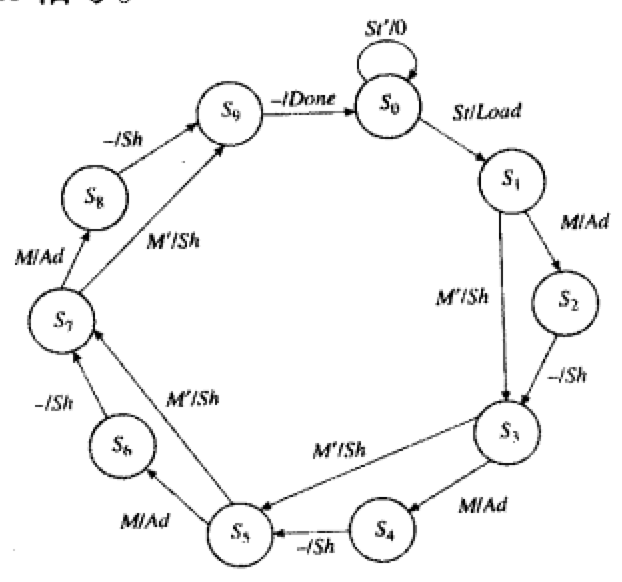


图 4.26　二进制乘法控制的状态图

行为描述方式 VHDL 程序（图 4.27）代码直接与状态图相对应。因为共有 10 状态，所以我们定义一个取值范围为 0～9 的整数作为状态信号。信号 ACC 表示 9 位的累加器输出。下面的语句

```
alias M: bit is ACC(0);
```

使我们可以用 *M* 代替 *ACC*(0)。语句“**when** 1|3|5|7 =>”表示当状态为 1, 3, 5, 7 时将执行语句后面的操作。所有的寄存器操作和状态改变都只在时钟的上升沿到来时发生。例如，在状态 S0，如果 *St* 为 1，则乘数被载入累加器，同时进入状态 S1。表达式'0'&ACC（7 **downto** 4）+Mcand 用于计算 2 个 4 位无符号矢量的和，并给出一个 5 位的结果。这一加法器的输出讲置入到 ACC 中去，它与状态计数器加 1 操作是同步进行。通过将 0 与 ACC 的高 8 位拼接并送入 ACC，来实现 ACC 的右移操作。表达式'0'&ACC（8 **downto** 1）可以用表达式 ACC **srl** 1 来代替。

```
-- This is a behavioral model of a multiplier for unsigned
-- binary numbers. It multiplies a 4-bit multiplicand
-- by a 4-bit multiplier to give an 8-bit product.

-- The maximum number of clock cycles needed for a
-- multiply is 10.

library IEEE;
use IEEE.numeric_bit.all;

entity mult4X4 is
  port(Clk, St: in bit;
       Mplier, Mcand: in unsigned(3 downto 0);
       Done: out bit);
end mult4X4;

architecture behave1 of mult4X4 is
signal State: integer range 0 to 9;
signal ACC: unsigned(8 downto 0); -- accumulator
alias M: bit is ACC(0);       -- M is bit 0 of ACC
begin
  process(Clk)
  begin
    if Clk'event and Clk = '1' then  -- executes on rising edge of clock
      case State is
        when 0 =>              -- initial State
          if St = '1' then
            ACC(8 downto 4) <= "00000"; -- begin cycle
            ACC(3 downto 0) <= Mplier;  -- load the multiplier
            State <= 1;
          end if;
```

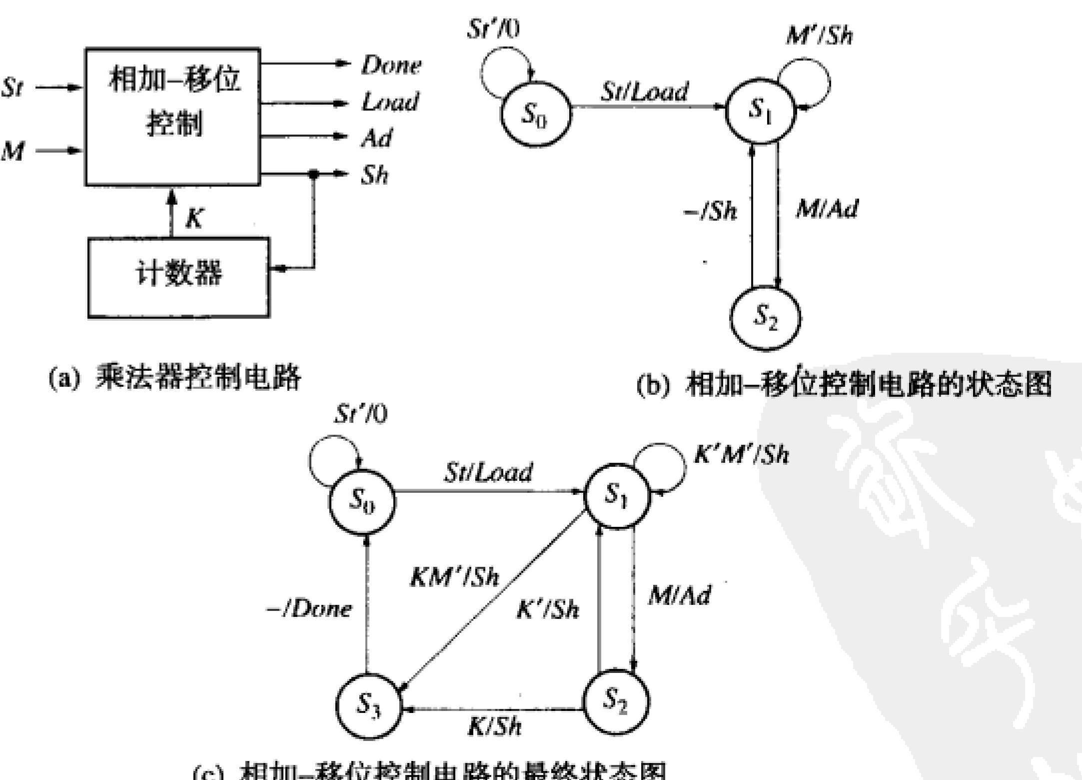
图 4.27　4×4 二进制乘法器的行为描述模块

```
        when 1 | 3 | 5 | 7 =>      -- "add/shift" State
          if M = '1' then          -- add multiplicand
            ACC(8 downto 4) <= '0' & ACC(7 downto 4) + Mcand;
            State <= State + 1;
          else
            ACC <= '0' & ACC(8 downto 1);       -- shift accumulator right
            State <= State + 2;
          end if;
        when 2 | 4 | 6 | 8 =>                   -- "shift" State
          ACC <= '0' & ACC(8 downto 1);         -- right shift
          State <= State + 1;
        when 9 =>                               -- end of cycle
          State <= 0;
      end case;
    end if;
  end process;
  Done <= '1' when State = 9 else '0';
end behave1;
```

图 4.27（续） 4×4 二进制乘法器的行为描述模块

完成信号(*Done*)需要在状态 S_9 时有效。如果我们使用语句 `when 9 => State<= 0;Done <='1'`，则 *Done* 信号将在状态变为 S_0 时才生效，这样就太晚了，因为我们需要在进入状态 S_9 时就发出 Done 信号。因此我们使用一条单独的并发赋值语句，放置于进程外，这样无论何时，只要状态发生改变，Done 信号就会得到更新。

乘法器的状态图（图 4.26）表明，控制执行有两功能——所需的相加或移位控制信号的生成和移位次数的计数。如果数字的位数很大，则把控制电路分成一个计数器和一个相加-移位控制器两部分就比较方便，如图 4.28(a)所示。首先，我们导出相加-移位控制器的状态图，用来测试 *St* 和 *M*，输出正确的相加和移位控制信号序列（图 4.28b）。然后，我们增加一个来自计数器的完成信号(*K*)，当完成正确次数的移位后，该信号停止乘法器工作。从图 4.28(b)的状态 S_0 开始，当收到开始信号 *St*=1 后，产生一个 *Load* 信号，电路进入状态 S_1。然后如果 *M*=1，则生成一个相加信号且电路进入状态 S_2；如果 *M*=0，则生成移位信号，且电路保持 S_1 状态。在状态 S_2 生成一个移位信号，因为在执行相加后总是要进行移位。图 4.28 (b)的状态图可以产生正确的相加和移位控制信号序列，但是不具有使乘法器停止工作的功能。



(a) 乘法器控制电路

(b) 相加–移位控制电路的状态图

(c) 相加–移位控制电路的最终状态图

图 4.28 带计数器的乘法器控制器

为了判断乘法运算何时结束，每成生一个移位信号，计数器都自动加 1。如果乘数是 *n* 位，则需进行 *n* 次移位。我们把计数器设计成当 *n*−1 次移位后，产生一个完成信号(*K*)。当 *K*=1 时，

如果需要，则电路应再做一次加法，然后做最后一次移位。只要 K=0，图 4.28(c)的控制操作与图 4.28(b)的一样。在状态 S_1，如果 K=1，则我们像平常一样检测 M。若 M=0，则输出最后一个移位信号并进入完成状态 S_3；若 M=1，则在移位前先进行加运算然后进入状态 S_2。在状态 S_2，若 K=1，则再输出一个移位信号，然后进入状态 S_3。最后一个移位将在相加-移位控制器进入完成状态的同时使计数器复位成零。

下面我们考虑图 4.25 的乘法器，但控制电路用图 4.28(a)来代替。因为 $n = 4$，所以需要一个 2 位的计数器来记录 4 次移位，且当计数器处于状态 3（11_2）时 K=1。表 4.2 给出了 1101 乘以 1011 时乘法器的运行情况。S_0, S_1, S_2, S_3 表示控制电路的状态（图 4.28c）。乘积寄存器每一步的内容与之前所给出的相同。

表 4.2 带计数器的乘法器的运行

时刻	状态	计数器	积寄存器	St	M	K	Load	Ad	Sh	Done
t_0	S_0	00	000000000	0	0	0	0	0	0	0
t_1	S_0	00	000000000	1	0	0	1	0	0	0
t_2	S_1	00	000001011	0	1	0	0	1	0	0
t_3	S_2	00	011011011	0	1	0	0	0	1	0
t_4	S_1	01	001101101	0	1	0	0	1	0	0
t_5	S_2	01	100111101	0	1	0	0	0	1	0
t_6	S_1	10	010011110	0	0	0	0	0	1	0
t_7	S_1	11	001001111	0	1	1	0	1	0	0
t_8	S_2	11	100011111	0	1	1	0	0	1	0
t_9	S_3	00	010001111	0	1	0	0	0	0	1

在 t_0 时刻，控制电路复位，等待开始信号。在 t_1 时刻，开始信号 St=1，生成 *Load* 信号。在 t_2 时刻 M=1，所以生成相加信号 Ad。当下一时钟信号出现时，加法器的输出被载入累加器，而控制系统进入状态 S_2。在 t_3 时刻，生成 Sh 信号，因此进行移位，且在下个时钟计数器加 1。在 t_4 时刻，M=1，所以 Ad=1，且加法器输出在下个时钟到来时被载入到累加器。在 t_5, t_6 时刻，移位并计数。在 t_7 时刻，已经进行了 3 次移位，计数器状态为 11，所以 K=1。由于 M=1，所以进行相加，控制系统进入状态 S_2。在 t_8 时刻，Sh=K=1，所以在下个时钟到来时进行最后一次移位，计数器加 1 回到 00 状态。在 t_9 时刻，生成 *Done* 信号。

只要通过增加寄存器的容量和计数器的位数，本节中给出的乘法器可以很容易的扩展成 8 位、16 位或者更多位数。而这些乘法器的相加-控制电路不变。

4.9 阵列结构乘法器

阵列结构乘法器是一种并行乘法器，它的部分积都是并行计算的。各个部分积在计算完毕后立即相加。此乘法器的计算过程如表 4.3 所示。表 4.3 演示了两个 4 位无符号数 $X_3X_2X_1X_0$ 和 $Y_3Y_2Y_1Y_0$ 的乘法运算过程，这两个数相乘可以得到一个 8 位的积。每个积 X_iY_j 均由一个与门生成。每个部分积都通过一行加法器与先前的部分积的和相加。第一行加法器计算头两行部分积的和，和输出为 $S_{13}S_{12}S_{11}S_{10}$，进位输出为 $C_{13}C_{12}C_{11}C_{10}$。其他两行加法器也可以得到相似的结果（我们用 S_{ij}, C_{ij} 来表示第 i 行加法器的和输入与进位输出）。

表 4.3 4 位乘法器分部积

				X_3	X_2	X_1	X_0	被乘数
				Y_3	Y_2	Y_1	Y_0	乘数
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	分部积 0
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		分部积 1
			C_{12}	C_{11}	C_{10}			第一行进位
		C_{13}	S_{13}	S_{12}	S_{11}	S_{10}		第一行和
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			分部积 2
		C_{22}	C_{21}	C_{20}				第二行进位
	C_{23}	S_{23}	S_{22}	S_{21}	S_{20}			第二行和
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				分部积 3
	C_{32}	C_{31}	C_{30}					第三行进位
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				第三行
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	最终积

图 4.29 给出了用与门和加法器阵列实现该乘法运算。如果一个加法器有三个输入，则使用全加器（FA），而如果一个加法器仅有两个输入，则使用半加器（HA）。若全加器的一个输入置零，则全加器与半加器相同。这种乘法器需要 16 个与门、8 个全加器和 4 个半加器。在输入 X, Y 后，进位必须在每行的各个单元间传递，而且和（*sum*）必须在行与行之间传输。完成乘法所需的时间主要由加法器的延迟决定。从输入到输出的最长路径经过 8 个加法器。如果 t_{ad} 是经由一个加法器所需的最大延迟，t_g 是最大的与门延迟，那么在最差情况下，完成乘法所需的时间为 $8t_{ad}+t_g$。

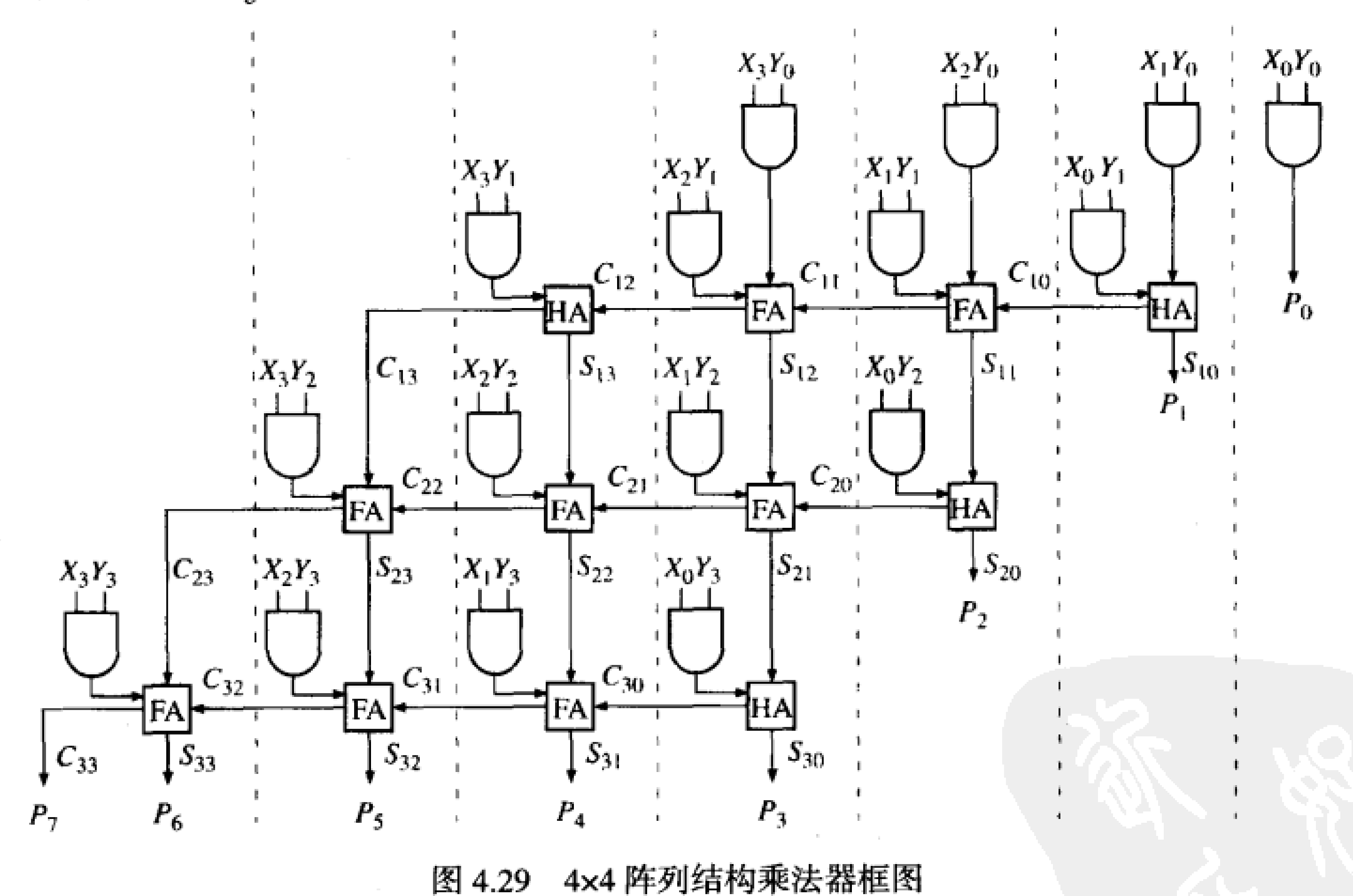


图 4.29 4×4 阵列结构乘法器框图

一般来说，一个 $n\times n$ 位的阵列结构乘法器需要 n^2 个与门、$n(n-2)$个全加器和 n 个半加器。因此阵列结构乘法器的组件数目随着 n 的增加而平方次增加。前面设计的串行-并行乘法器的控制电路所需的硬件数目是随着 n 线性增加的。

前面设计的这类串行-并行结构乘法器，在最差情况下需要 $2n$ 个时钟来完成乘法运算。我们可以通过下一节介绍的技术使此时间减少至 n 个时钟。时钟最小周期取决于 n 位加法器的传输延迟、累加触发器的传输延迟和建立时间。

4.9.1 VHDL 编程

如果设计者要求特定的拓扑结构，则必须采用结构描述的 VHDL 编程，如图 4.30 所示。如果不考虑特定的拓扑结构，而使用行为描述方式编程，那么综合器生成的拓扑结构将取决于综合工具。这里，我们给出了阵列结构乘法器的结构描述 VHDL 代码。全加器和半加器模块作为组成元件用于阵列结构乘法器中，它们根据阵列结构乘法器的拓扑结构进行连接，因此这里我们使用了元件例化语句（Port Map）。

```
entity Array_Mult is
  port(X, Y: in bit_vector(3 downto 0);
       P: out bit_vector(7 downto 0));
end Array_Mult;

architecture Behavioral of Array_Mult is
signal C1, C2, C3: bit_vector(3 downto 0);
signal S1, S2, S3: bit_vector(3 downto 0);
signal XY0, XY1, XY2, XY3: bit_vector(3 downto 0);
component FullAdder
  port(X, Y, Cin: in bit;
       Cout, Sum: out bit);
end component;
component HalfAdder
  port(X, Y: in bit;
       Cout, Sum: out bit);
end component;
begin
  XY0(0) <= X(0) and Y(0);   XY1(0) <= X(0) and Y(1);
  XY0(1) <= X(1) and Y(0);   XY1(1) <= X(1) and Y(1);
  XY0(2) <= X(2) and Y(0);   XY1(2) <= X(2) and Y(1);
  XY0(3) <= X(3) and Y(0);   XY1(3) <= X(3) and Y(1);

  XY2(0) <= X(0) and Y(2);   XY3(0) <= X(0) and Y(3);
  XY2(1) <= X(1) and Y(2);   XY3(1) <= X(1) and Y(3);
  XY2(2) <= X(2) and Y(2);   XY3(2) <= X(2) and Y(3);
  XY2(3) <= X(3) and Y(2);   XY3(3) <= X(3) and Y(3);
  FA1: FullAdder port map (XY0(2), XY1(1), C1(0), C1(1), S1(1));
  FA2: FullAdder port map (XY0(3), XY1(2), C1(1), C1(2), S1(2));
  FA3: FullAdder port map (S1(2), XY2(1), C2(0), C2(1), S2(1));
  FA4: FullAdder port map (S1(3), XY2(2), C2(1), C2(2), S2(2));
  FA5: FullAdder port map (C1(3), XY2(3), C2(2), C2(3), S2(3));
  FA6: FullAdder port map (S2(2), XY3(1), C3(0), C3(1), S3(1));
  FA7: FullAdder port map (S2(3), XY3(2), C3(1), C3(2), S3(2));
  FA8: FullAdder port map (C2(3), XY3(3), C3(2), C3(3), S3(3));
  HA1: HalfAdder port map (XY0(1), XY1(0), C1(0), S1(0));
  HA2: HalfAdder port map (XY1(3), C1(2), C1(3), S1(3));
  HA3: HalfAdder port map (S1(1), XY2(0), C2(0), S2(0));
```

图 4.30 4×4 阵列结构乘法器的 VHDL 代码

```
 HA4: HalfAdder port map (S2(1), XY3(0), C3(0), S3(0));

 P(0) <= XY0(0);  P(1) <= S1(0);  P(2) <= S2(0);
 P(3) <= S3(0);  P(4) <= S3(1);  P(5) <= S3(2);
 P(6) <= S3(3);
 P(7) <= C3(3);
end Behavioral;

-- Full Adder and half adder entity and architecture descriptions
-- should be in the project
entity FullAdder is
 port(X, Y, Cin: in bit;
      Cout, Sum: out bit);
end FullAdder;

architecture equations of FullAdder is
begin
 Sum <= X xor Y xor Cin;
 Cout <= (X and Y) or (X and Cin) or (Y and Cin);
 end equations;
entity HalfAdder is
 port(X, Y: in bit;
      Cout, Sum: out bit);
end HalfAdder;

architecture equations of HalfAdder is
begin
 Sum <= X xor Y;
 Cout <= X and Y;
end equations;
```

图 4.30（续）　4×4 阵列结构乘法器的 VHDL 代码

4.10　有符号整数/分数的乘法

二进制有符号数乘法的可行算法有好几种，下面介绍的是一种直接计算方法：

1. 如果乘数为负，则取补。
2. 如果被乘数为负，则取补。
3. 对两个正的二进制数做乘法。
4. 如果乘积应该为负，则对积取补。

尽管上面的方法在概念上很简单，但是与其他方法相比，它需要较多的硬件和计算时间。

下面介绍另一种方法，它仅需要对被乘数取补，对乘数和积不必求补。这种方法对于整数和小数都同样适用。由于接下来我们会用这种乘法器作为浮点数乘法器的一个组成部分，所以下面我们用小数来进行举例说明。用二进制补码表示负数，则有符号二进制小数表示如下：

0.101　+5/8　　1.011　−5/8

二进制小数点左边的数是符号位，0 表示正小数，1 表示负小数。一般来说，二进制小数 F 的补码 $F^*=2-F$。所以，10.000–0.101=1.011 表示–5/8（这种定义小数二进制补码的方法与整数情况相同($N^*=2^n-N$)，因为将小数点左移 $n-1$ 位相当于除以 2^{n-1}）。小数的取补操作可以从最右边位开始，将第一个 1 左边的所有位取反，整数取补运算与此类似。1.000 的补码是特殊情况，它通常用来表示–1。由于符号位为负，所以 1.000…的二进制补码为 2–1=1。 由于 0.111…是最大的正小数，所以在二进制补码小数系统中不能表示+1。

二进制定点小数

定点数是一种数据格式，其十进制或二进制数的小数点在固定的地方。我们可以设置一个定点 8 位数，二进制小数点在第 4 位之后，即 4 位是小数，4 位为整数。如果二进制小数点再向右移两位，则有 6 位为整数，2 位为小数。数的范围和精度随着小数点位置的不同而不同。例如，若 4 位是小数，4 位为整数，则此无符号数的范围为 0.00～15.925；若 6 位为整数，2 位为小数，则此数的范围为 0.00～63.75，但是小数部分只可以精确到 0.25。可见，其范围增加了，但是精度却降低了。

下面我们使用二进制补码定点数表示–13.45，小数位为 4 位。在把十进制小数转换为二进制小数的时候，我们只需把小数部分（注意只是小数部分）反复地乘以 2，所以我们从 0.45 开始，每次乘以 2，计算过程为

0.90
1.80
1.60
1.20
0.40
0.80
1.60
1.20

现在小数部分的二进制表示就可以用上面加黑的数字来表示。注意，根据不同的小数部分位数，我们选择要使用多少位（例如，小数位数为 4 时选择 0111；小数位数为 8 时选择 01110011 等）。十进制数 13.45 的二进制表示为（小数部分为 4 位）

13.45:　　1101.0111

注意，此二进制表示只是原数的近似值，若把其转换为十进制数，我们得到 13.4375（与我们的初始数据有细微的不同）。当可使用的二进制数位数越多时，转换后的数与原数据就越接近。

负小数可以用二进制补码的形式表示。下面我们把–13.45 表示成二进制补码形式。如果整数位只有 4 位，那么此转换将无法进行，所以我们至少使用 5 位整数位表示这个有符号数。假设整数位为 5 位，则转换后的二进制数一共有 9 位：

13.45:　　01101.0111
反码：　　10010.1000
补码：　　10010.1001

因此，在二进制定点数格式中–13.45 = 10010.1001。

当进行有符号数二进制数的乘法运算时，我们必须考虑以下四种情况：

被乘数	乘数
+	+
−	+
+	−
−	−

当被乘数和乘数均为正时，使用标准的二进制乘法。例如，

运算	值		说明
0.1 1 1	(+7/8)	←	被乘数
× 0.1 0 1	(+5/8)	←	乘数
(0. 0 0)0 1 1 1	(+7/64)	←	注意：如果要正确的表示小数部分积，那么就需
(0.)0 1 1 1	(+7/16)	←	要越过小数点对符号位进行扩展，如括号中所示
0. 1 0 0 0 1 1	(+35/64)		（在硬件中无需进行符号扩展）

当被乘数为负、乘数为正时，操作步骤与前边相同，只需扩展被乘数的符号位，这样可保证部分积和最终积的符号正确。例如，

运算	值		说明
1.1 0 1	(−3/8)		
× 0.1 0 1	(+5/8)		
(1. 1 1)1 1 0 1	(−3/64)	←	注意：符号位扩展使负积
(1.)1 1 0 1	(−3/16)	←	可以正确的表达
1. 1 1 0 0 0 1	(−15/64)		

当乘数为负、被乘数为正时，我们必须把乘法运算过程做一些变动。1.g 形式的负小数在数值上可以表示为−1+0.g; 例如，1.011= −1+0.011= −(1−0.011)= −0.101= −5/8。这样，当乘以一个 1.g 形式的负小数时，把小数部分(.g)按照正小数进行运算，但是符号位为−1。这样，我们就可以按照正常的步骤进行乘法计算，把被乘数乘以小数的每一位，并把部分积累加起来。然而，当计算到负的符号位时，我们必须加上被乘数的补码，而不是被乘数本身。我们用下面例子进行说明：

运算	值		说明
0.1 0 1	(+5/8)		
× 1.1 0 1	(−3/8)		
(0. 0 0)0 1 0 1	(+5/64)		
(0.)0 1 0 1	(+5/16)		
(0.)0 1 1 0 0 1			
1. 0 1 1	(−5/8)	←	注意：在此点加上被乘数的二进制补码
1. 1 1 0 0 0 1	(−15/64)		

当被乘数和乘数都是负数的时候，乘法运算过程是与前面是一样的。在每一步，必须小心地扩展部分积的符号位，以保持正确的符号，且最后一步，由于乘数的符号位是负的，所以要用被乘数的二进制补码形式进行加操作。例如，

运算	值		说明
1.1 0 1	(−3/8)		
× 1.1 0 1	(−3/8)		
(1. 1 1)1 1 0 1	(−3/64)	←	注意：扩展符号位
(1.)1 1 0 1	(−3/16)		
1. 1 1 0 0 0 1			
0. 0 1 1	(+3/8)	←	注意：在此点加上被乘数的二进制补码
0. 0 0 1 0 0 1	(+9/64)		

总之，有符号二进制补码小数乘法的计算过程与二进制正小数乘法的计算过程是相同的，只

是在每一步均需要注意保持部分积的符号，且如果乘数的符号为负时，在最后一步对被乘数进行加运算前，需要对被乘数取补。二进制补码小数乘法的乘法器的硬件实现与正数乘法几乎是一样的，只是要增加一个被乘数取补器。

图 4.31 给出了实现两个 4 位小数（包括符号位）的乘法运算所需的硬件。由于进位会占据符号位，所以为了保证和的符号不丢失，就需要一个 5 位的加法器。控制器的输入 M 是当前正在计算的乘数位。控制信号 Sh 可以使累加器和符号扩展位整体右移一位。信号 Ad 会使输出 ADDER 被读入累加器的左 5 位。由于我们要进行二进制补码加法，所以舍掉加法器的最后一位的进位输出。Cm 使被乘数(Mcand)在进入加法器输入前取反（二进制反码）。同时，Cm 与加法器的进位输入相连，所以当 Cm=1，我们将把 Mcand 的反码加上 1，然后再加到累加器中，这就相当于将被乘数的补码加入累加器。图 4.32 给出了控制电路的状态图。我们对每一个乘数位(M)都进行测试，以确定是进行相加移位，还是仅移位。在状态 S_7、M 为符号位，如果 M=1，则将被乘数的补码加到累加器。

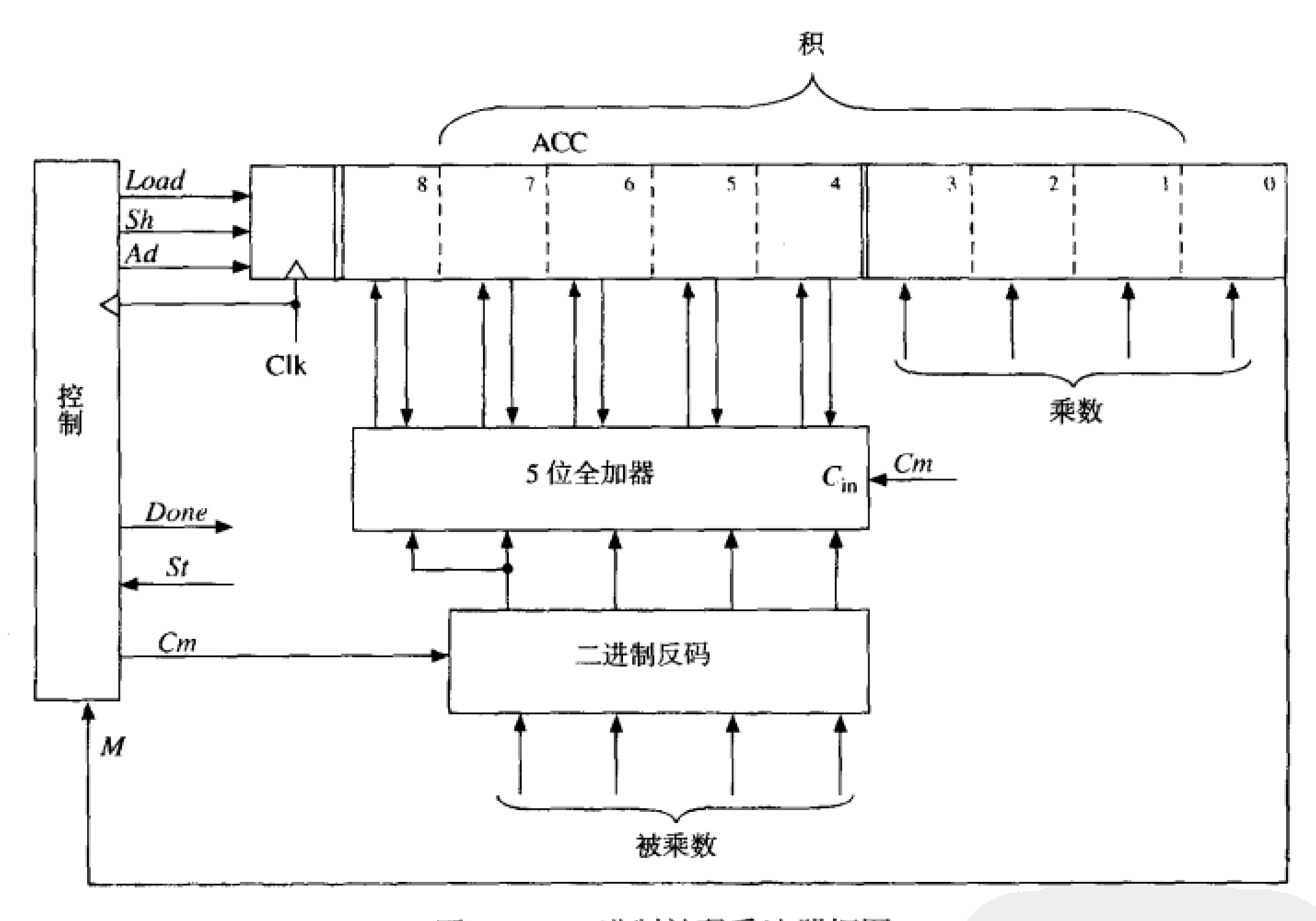


图 4.31　二进制补码乘法器框图

当使用图 4.31 中的硬件时，相加和移位操作必须在两个单独的时钟进行。我们可以加快乘法器的操作速度，只要将加法器输出线路最右边的一条线再右移一位（图 4.33），这样当加法器输出被读入到累加器的时候，相当于已经移了一位。因此，图 4.33 所示乘法器的相加和移位操作可以在一个时钟内进行。其控制器的状态图见图 4.34。当乘法完成时，积（6 位再加上符号位）位于 A 的低 3 位和 B 中。积的小数点位于寄存器 A 的中间位置。如果想让它位于左边两位之间，就要对 A 和 B 整体左移一位。

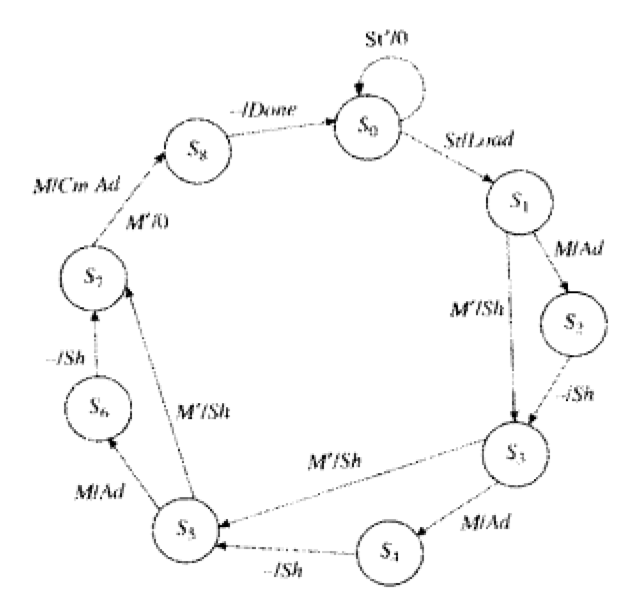


图 4.32　二进制补码乘法器的状态图

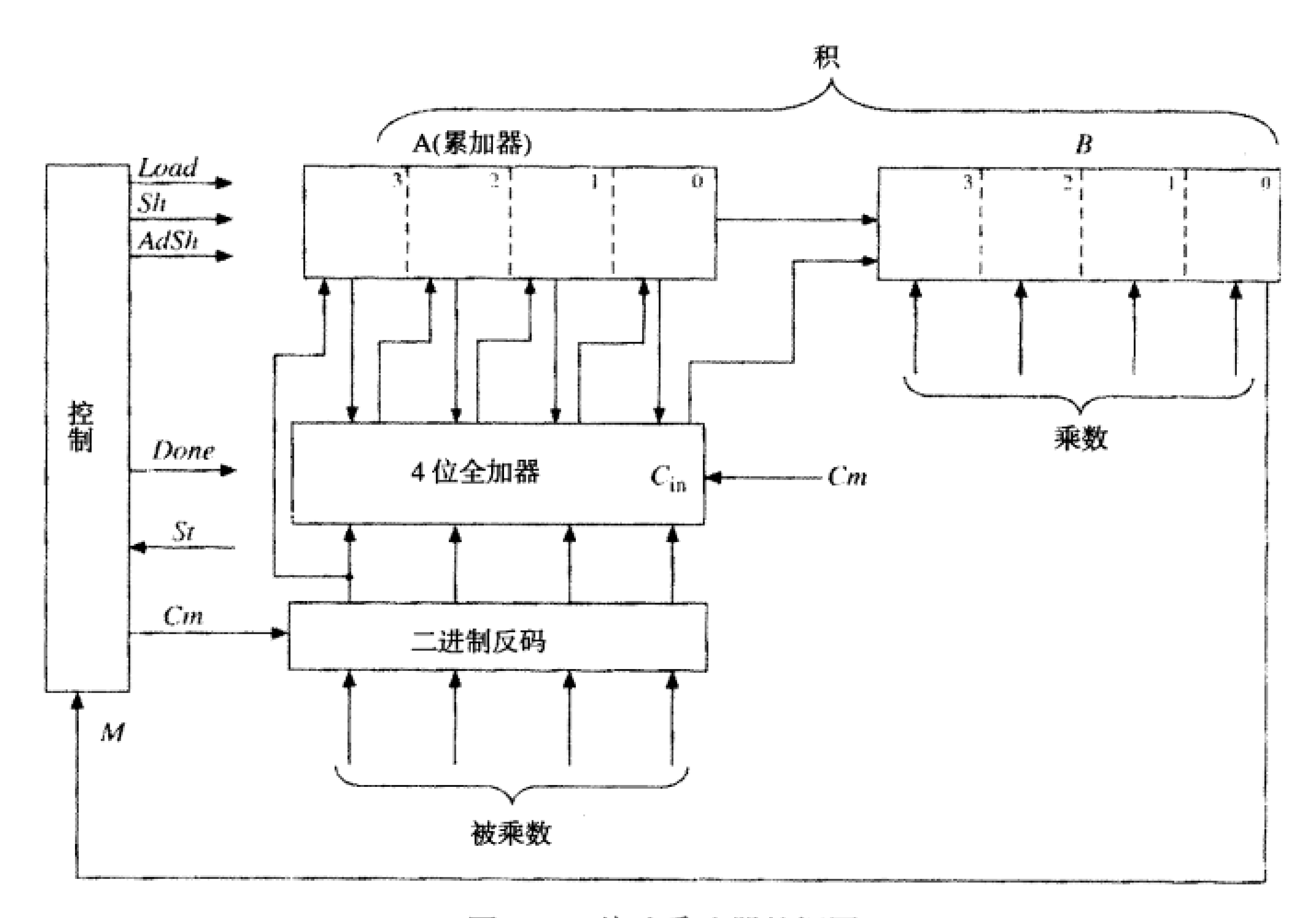


图 4.33　快速乘法器的框图

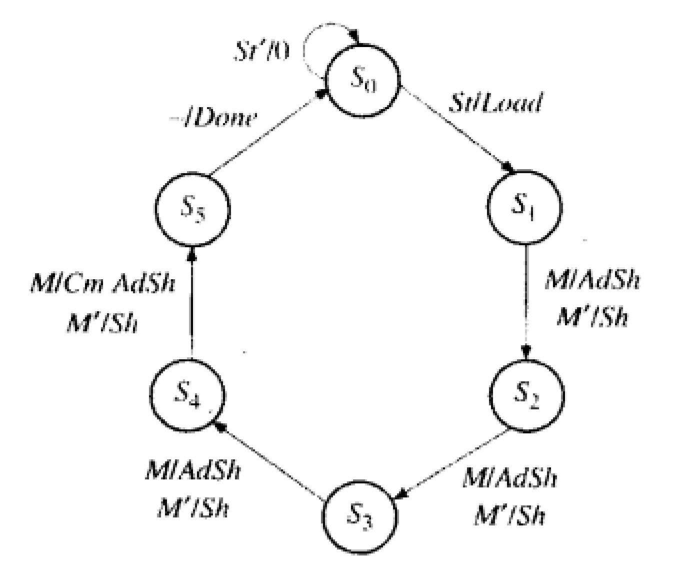


图 4.34　快速乘法器的状态图

如图 4.35 展示了此乘法器的行为描述方式 VHDL 模块。对寄存器 A 和 B 进行整体移位是用下面的顺序语句实现的：

```
A <= A(3) & A (3 downto 1);
B <= A(0) & B (3 downto 1);
```

尽管这些语句是按顺序执行的，但是 A 和 B 应该在同一个 Δ 时间内更新，因此 $A(0)$的旧值用来计算 B 的新值。

```
library IEEE;
use IEEE.numeric_bit.all;

entity mult2C is
  port(CLK, St: in bit;
       Mplier, Mcand : in unsigned(3 downto 0);
       Product: out unsigned (6 downto 0);
       Done: out bit);
end mult2C;

architecture behave1 of mult2C is
signal State: integer range 0 to 5;
signal A, B: unsigned(3 downto 0);
alias M: bit is B(0);
begin
  process(CLK)
  variable addout: unsigned(3 downto 0);
  begin
    if CLK'event and CLK = '1' then
      case State is
        when 0 =>                      -- initial State
          if St = '1' then
            A <= "0000";               -- begin cycle
            B <= Mplier;               -- load the multiplier
            State <= 1;
          end if;
        when 1 | 2 | 3 =>              -- "add/shift" states
          if M = '1' then
            addout := A + Mcand;    -- add multiplicand to A and shift
            A <= Mcand(3) & addout(3 downto 1);
            B <= addout(0) & B(3 downto 1);
          else
            A <= A(3) & A(3 downto 1);  -- arithmetic right shift
            B <= A(0) & B(3 downto 1);
          end if;
          State <= State + 1;
        when 4 =>
          if M = '1' then
            addout := A + not Mcand + 1;
              -- add 2's complement when sign bit of multiplier is 1
            A <= not Mcand(3) & addout(3 downto 1);
            B <= addout(0) & B(3 downto 1);
          else
            A <= A(3) & A(3 downto 1);  -- arithmetic right shift
            B <= A(0) & B(3 downto 1);
          end if;
          State <= 5;
        when 5 =>
          State <= 0;
      end case;
    end if;
  end process;
  Done <= '1' when State = 5 else '0';
  Product <= A(2 downto 0) & B;     -- output product
end behave1;
```

图 4.35 二进制补码乘法器的行为描述模块

变量 *addout* 用来表示加法器的 5 位输出。从状态 1 到状态 4，如果当前的乘数位 M 为'1'，

那么被乘数的符号和 *addout* 的高3位被载入到 *A* 中。同时，*addout* 的第4位和 *B* 的高三位被载入到 *B* 中。当控制器进入状态5时，生成 *Done* 信号，然后新计算出的积的值被输出。

在继续这个设计之前，我们要对行为描述方式的VHDL代码进行测试，以保证算法的正确性和同硬件框图的一致性。在测试的初期阶段，我们想要通过单步输出来检验乘法器的内部操作，如果需要可以进行调试。当我们认为乘法器工作正常后，就会只关注它的最终输出结果（积），这样就可以快速的对大量的情况进行测试。

图4.36给出了命令文件，以及+5/8与−3/8相乘的测试结果。时钟周期为20 ns，*St* 信号在2 ns时启动，在一个时钟周期后撤销。通过对状态图的监测，我们知道此乘法操作需要6个时钟，所以整个运行时间设置为120 ns。

```
-- command file to test signed multiplier
add list CLK St State A B Done Product
force st 1 2, 0 22
force clk 1 0, 0 10 - repeat 20
-- (5/8 * -3/8)
force Mcand 0101
force Mplier 1101
run 120
```

ns	delta	CLK	St	State	A	B	Done	Product
0	+1	1	0	0	0000	0000	0	0000000
2	+0	1	1	0	0000	0000	0	0000000
10	+0	0	1	0	0000	0000	0	0000000
20	+1	1	1	1	0000	1101	0	0000000
22	+0	1	0	1	0000	1101	0	0000000
30	+0	0	0	1	0000	1101	0	0000000
40	+1	1	0	2	0010	1110	0	0000000
50	+0	0	0	2	0010	1110	0	0000000
60	+1	1	0	3	0001	0111	0	0000000
70	+0	0	0	3	0001	0111	0	0000000
80	+1	1	0	4	0011	0011	0	0000000
90	+0	0	0	4	0011	0011	0	0000000
100	+2	1	0	5	1111	0001	1	1110001
110	+0	0	0	5	1111	0001	1	1110001
120	+1	1	0	0	1111	0001	0	1110001

图4.36 命令文件和（+5/8乘以−3/8）的仿真结果

为了全面测试这个乘法器，不但要测试四种标准情况(++，+−，−+和−−)，还要针对一些特殊的情况和受限的情况进行测试。被乘数和乘数的测试值应该包括0、最大的正小数、最小的负小数和全1。下面我们设计一个VHDL测试平台对此乘法器进行测试。这个测试平台提供了一个被乘数和乘数的测试值序列，这样就为此乘法器的测试系统提供了一个激励系列。此测试平台也可以对乘法器输出结果的正确性进行测试。我们把乘法器嵌入到测试平台中，作为测试平台的一个元件进行测试。测试平台生成的，用于乘法器的接口信号见图4.37。

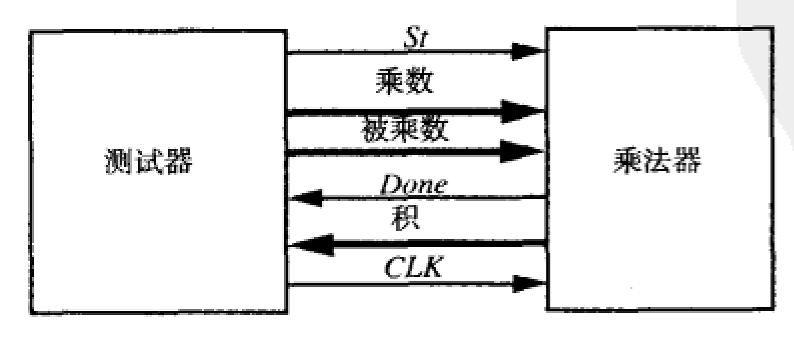


图4.37 乘法器及其测试平台之间的接口

图4.38给出了乘法器测试平台的VHDL代码。测试序列由11对被乘数和乘数构成，分别存储于两个数组 *Mcandarr* 和 *Mplierarr* 中。为了测试乘法器输出的正确性，我们把所有期望输出存

储于另一个数组 *Productarr* 中。在 VHDL 代码中，测试值及其结果均放在常数数组中。我们把乘法器作为测试平台的一个元件，所以使用了 **port map** 语句引用了该乘法器元件。该测试平台还可以产生时钟信号和开始信号。for 循环语句从 *Mcandarr* 和 *Mplierarr* 数组中读取测试值，然后置开始信号为 1，在下一个时钟又关掉开始信号（置为 0）。随后测试平台等待 *Done* 信号的到来。当 *Done* 信号的触发沿到来时，将乘法器的输出同数组 *Productarr* 中存储的期望输出进行比较。如果不匹配，就报错。当乘法器的控制回到 S_0 状态时候，就关掉 *Done* 信号。因此，在 *done* 信号的下降沿到来之后，才提供新的被乘数(*Mcand*)和乘数(*Mplier*)用于再次循环计算。注意到 **port map** 语句在产生激励进程的外边。乘法器一直不断接收一组输入数据，并生成相应的一组输出数据。

```
library IEEE;
use IEEE.numeric_bit.all;

entity testmult is
end testmult;

architecture test1 of testmult is
component mult2C
  port(CLK, St: in bit;
       Mplier, Mcand: in unsigned(3 downto 0);
       Product: out unsigned(6 downto 0);
       Done: out bit);
end component;

constant N: integer := 11;
type arr is array(1 to N) of unsigned(3 downto 0);
type arr2 is array(1 to N) of unsigned(6 downto 0);
constant Mcandarr: arr := ("0111", "1101", "0101", "1101", "0111",
                   "1000", "0111", "1000", "0000", "1111", "1011");
constant Mplierarr: arr := ("0101", "0101", "1101", "1101", "0111",
                    "0111", "1000", "1000", "1101", "1111", "0000");
constant Productarr: arr2 := ("0100011", "1110001", "1110001",
                              "0001001", "0110001", "1001000",
                              "1001000", "1000000", "0000000",
                              "0000001", "0000000");
signal CLK, St, Done: bit;
signal Mplier, Mcand: unsigned(3 downto 0);
signal Product: unsigned(6 downto 0);
begin
  CLK <= not CLK after 10 ns;
  process
  begin
    for i in 1 to N loop
      Mcand <= Mcandarr(i);
      Mplier <= Mplierarr(i);
      St <= '1';
      wait until CLK = '1' and CLK'event;
      St <= '0';
      wait until Done = '0' and Done'event;
      assert Product = Productarr(i)    -- compare with expected answer
        report "Incorrect Product"
        severity error;
    end loop;
    report "TEST COMPLETED";
  end process;
  mult1: mult2c port map(CLK, St, Mplier, Mcand, Product, Done);
end test1;
```

图 4.38　有符号乘法器的测试平台

图 4.39 给出了命令文件和仿真器输出。我们已经对仿真器输出进行了评注，以解释测试的结果。列表中–NOtrigger 和 Trigger 使得电路仅在 *Done* 发生改变的时候才有输出显示。如果没有 –NOtrigger 和 Trigger，那么当列表中任意一个信号发生变化时，就会有输出。除了

−1×−1(1.000×1.000)这种特殊情况外，其他乘积输出结果都是正确的。对于−1×−1，输出结果为 1.000000(−1)，而不是+1。这是因为如果没有增加位数，那么无法表示+1。

接下来，我们通过对控制信号和每个控制信号引起的操作进行明确定义，进而对有符号数乘法器的 VHDL 模块修改优化。我们使用与图 1.17 中的 Mealy 机相似的结构编写 VHDL 代码（图 4.40）。在第一个进程中，我们对每个当前状态(*State*)的下一个状态(*Nextstate*)和输出控制信号进行了定义。在第二个进程中，当时钟上升沿到来后，更新了相应的寄存器以更新状态。我们可以用前面用过的测试文件对图 4.40 中的 VHDL 代码进行测试，检验它是否能够得到相同的乘积输出。

```
-- Command file to test results of signed multiplier
add list -NOtrigger Mplier Mcand product -Trigger done
run 1320

  ns    delta    mplier    mcand    product    done
   0      +1      0101      0111    0000000      0
  90      +2      0101      0111    0100011      1     5/8 * 7/8 = 35/64
 110      +2      0101      1101    0100011      0
 210      +2      0101      1101    1110001      1     5/8 * -3/8 = -15/64
 230      +2      1101      0101    1110001      0
 330      +2      1101      0101    1110001      1     -3/8 * 5/8 = -15/64
 350      +2      1101      1101    1110001      0
 450      +2      1101      1101    0001001      1     -3/8 * -3/8 = 9/64
 470      +2      0111      0111    0001001      0
 570      +2      0111      0111    0110001      1     7/8 * 7/8 = 49/64
 590      +2      0111      1000    0110001      0
 690      +2      0111      1000    1001000      1     7/8 * -1 = -7/8
 710      +2      1000      0111    1001000      0
 810      +2      1000      0111    1001000      1     -1 * 7/8 = -7/8
 830      +2      1000      1000    1001000      0
 930      +2      1000      1000    1000000      1     -1 * -1 = -1 (error)
 950      +2      1101      0000    1000000      0
1050      +2      1101      0000    0000000      1     -3/8 * 0 = 0
1070      +2      1111      1111    0000000      0
1170      +2      1111      1111    0000001      1     -1/8 * -1/8 = 1/64
1190      +2      0000      1011    0000001      0
1290      +2      0000      1011    0000000      1     0 * -3/8 = 0
1310      +2      0101      0111    0000000      0
```

图 4.39　有符号乘法器的命令文件及仿真结果

```
-- This VHDL model explicitly defines control signals.

library IEEE;
use IEEE.numeric_bit.all;
entity mult2C is
  port(CLK, St: in bit;
       Mplier, Mcand: in unsigned(3 downto 0);
       Product: out unsigned (6 downto 0);
       Done: out bit);
end mult2C;

-- This architecture of a 4-bit multiplier for 2's complement numbers
-- uses control signals.

architecture behave2 of mult2C is
signal State, Nextstate: integer range 0 to 5;
signal A, B, compout, addout: unsigned(3 downto 0);
signal AdSh, Sh, Load, Cm: bit;
alias M: bit is B(0);
begin
  process(State, St, M)
  begin
    Load <= '0'; AdSh <= '0'; Sh <= '0'; Cm <= '0'; Done <= '0';
    case State is
```

图 4.40　带有控制信号的二进制补码乘法器模块

```
      when 0 =>              -- initial state
        if St = '1' then Load <= '1'; Nextstate <= 1; end if;
      when 1 | 2 | 3 = >    -- "add/shift" State
        if M = '1' then AdSh < = '1';
        else Sh <= '1';
        end if;
         Nextstate <= State + 1;
      when 4 =>              -- add complement if sign
         if M = '1' then     -- bit of multiplier is 1
           Cm <= '1'; AdSh <= '1';
         else Sh <= '1';
         end if;
         Nextstate <= 5;
      when 5 =>              -- output product
         Done <= '1';
         Nextstate <= 0;
    end case;
  end process;

  compout <= not Mcand when Cm = '1' else Mcand;  -- complementer
  addout <=  A + compout + unsigned'(0=>Cm);      -- 4-bit adder with carry in

  process (CLK)
  begin
    if CLK'event and CLK = '1' then   -- executes on rising edge
      if Load = '1' then              -- load the multiplier
        A <= "0000";
        B <= Mplier;
      end if;
      if AdSh = '1' then              -- add multiplicand to A and shift
        A <= compout(3) & addout(3 downto 1);
        B <= addout(0) & B(3 downto 1);
      end if;
      if Sh = '1' then
        A <= A(3) & A(3 downto 1);
        B <= A(0) & B(3 downto 1);
      end if;
      State <= Nextstate;
    end if;
  end process;
  Product <= A(2 downto 0) & B;
end behave2;
```

图 4.40（续） 带有控制信号的二进制补码乘法器模块

4.11 键盘扫描器

在本节，我们将为一个 4 行 3 列的键盘设计一个扫描器，键盘如图 4.41 所示。键区是按照矩阵形式布线的，在每行和每列的交叉处都有一个开关。每按一个键就代表在行和列之间建立一个连接。扫描器的作用是确定哪一个键被按下并输出一个与键号相符的二进制数 $N=N_3N_2N_1N_0$。例如，按键 5 将输出 0101，按 * 键将输出 1010，按 # 键将输出 1011。当检测到一个有效按键时，扫描器应输出一个时钟时间的信号 V。我们假设每次只有一个键被按下。该设计中必须包含一个硬件以保护电路避免由键盘抖动引起的误操作。

1	2	3
4	5	6
7	8	9
*	0	#

图 4.41 4 行 3 列式键盘

电路的整体框图见图 4.42 所示。键盘中含有接地电阻。每当按下一个开关时，就会在对应的

列与地面之间形成一条通路。如果在列线 C_2, C_1 和 C_0 上加载电压，则按键后，在相应的行线上就可以得到该电压，行 R_0, R_1, R_2 或 R_3 中的一个就会存在有效信号。

我们把该设计分成几个模块，如图 4.43 所示。设计的第一部分为键盘行列扫描器。键盘扫描模块产生成列信号用于扫描键盘。当某个键被按下时，去抖动模块产生一个信号 K，去抖动后生成信号 Kd。当检测到一个有效的按键时，译码器根据其所在的行和列确定该键号。

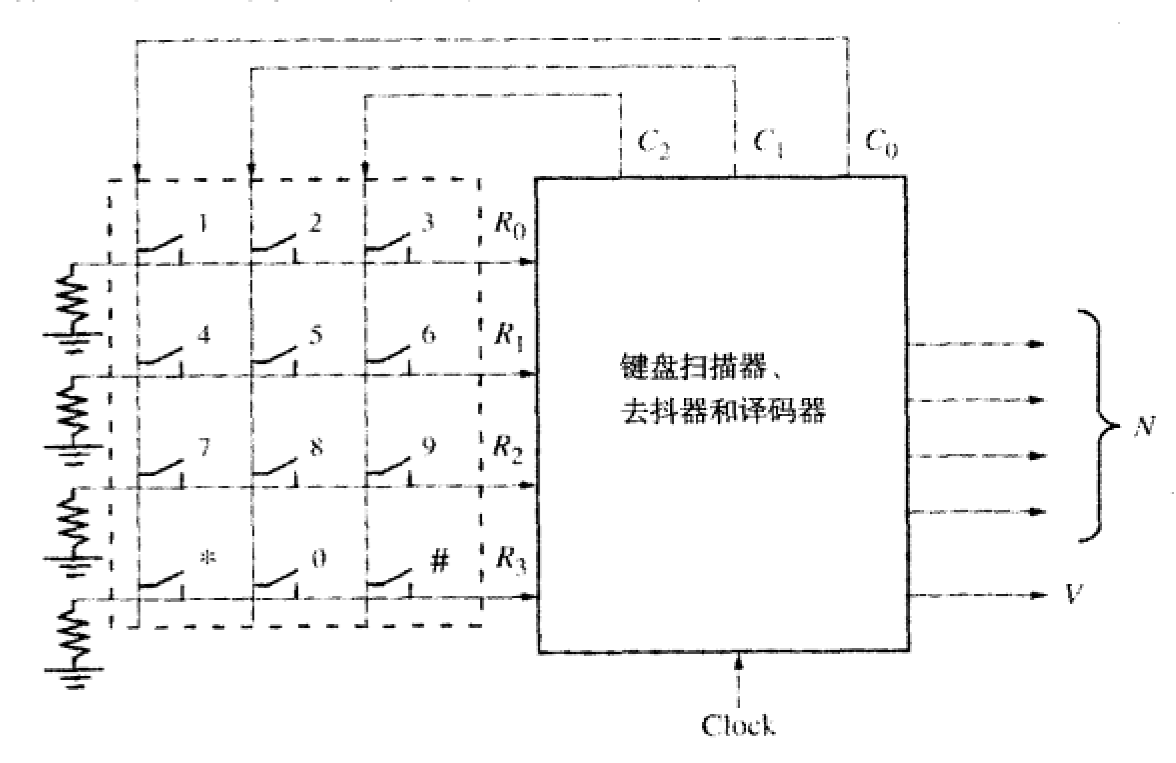


图 4.42 键盘扫描器的框图

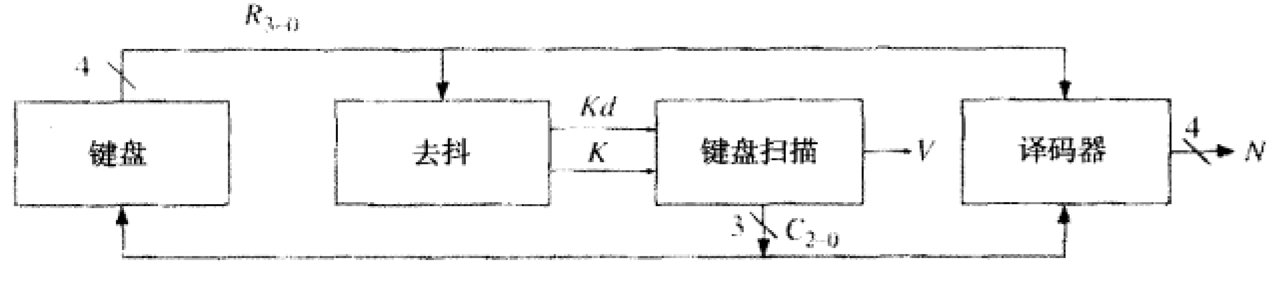


图 4.43 扫描器模块

4.11.1 扫描器

我们按照下面的步骤进行键盘扫描：首先将 C_2, C_1 和 C_0 置为逻辑 1 并等待。如果有键被按下，R_0, R_1, R_2 或 R_3 中将有一个会为 1。然后只将列 C_0 置 1，如果 R_i 中有任意一个为 1，则说明检测到一个有效的按键。如果接收到 R_0，则键 1 被按下；如果 R_1, R_2 或 R_3 被接收到，则键 4, 7 或*键被按下。此时把 V 置为 1 并输出相应的 N。如果没有在第一列中检测到按键，则把 C_1 置 1 并重复以上过程；如果没有在第二列检测到按键，则把 C_2 置 1 并重复以上过程。当检测到一个键被按下时，依次把 C_2, C_1 或 C_0 置 1，直到再没有键被按下为止。最后这个步骤是必要的，这样当每次一个键被按下后，就只会生成一个有效信号。

4.11.2 去抖动器

和记分板例子一样，为了防止由于开关抖动而引起的误操作，我们也需要按键的去抖动。图 4-44 给出了一个去抖动和同步电路。4 个行信号与一个或门相连产生信号 K。当有一个键被按下时，信号 K 就有效，列扫描信号就产生，去抖动的信号 Kd 将馈给时序电路的输入端。

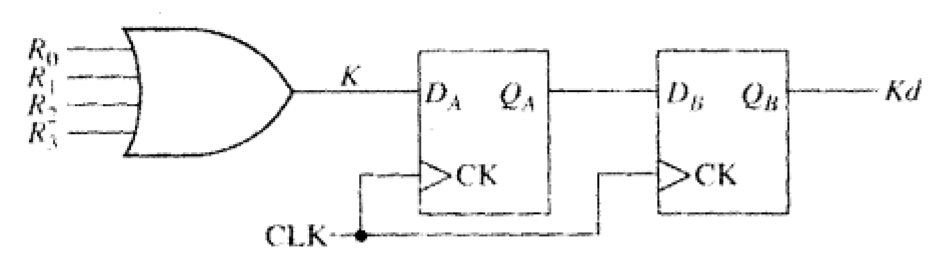


图 4.44 去抖动和同步电路

4.11.3 译码器

译码器通过行号和列号来决定键号，译码器所使用的真值表见表 4.4。真值表的每一行代表 12 个键中的一个。由于我们假设每次只有一个键被按下，所以剩下的行的输出都为随意项。 因为译码器是一个组合电路，所以它的输出会在键盘扫描中改变。当检测到一个有效的按键时（K=1, V=1），它输出正确的值，并且这个值在电路进入状态 S_5 的同时被存入寄存器中。

表 4.4 译码器的真值表

R_3	R_2	R_1	R_0	C_0	C_1	C_2	N_3	N_2	N_1	N_0	
0	0	0	1	1	0	0	0	0	0	1	
0	0	0	1	0	1	0	0	0	1	0	
0	0	0	1	0	0	1	0	0	1	1	
0	0	1	0	1	0	0	0	1	0	0	
0	0	1	0	0	1	0	0	1	0	1	
0	0	1	0	0	0	1	0	1	1	0	
0	1	0	0	1	0	0	0	1	1	1	
0	1	0	0	0	1	0	1	0	0	0	
0	1	0	0	0	0	1	1	0	0	1	
1	0	0	0	1	0	0	1	0	1	0	(*)
1	0	0	0	0	1	0	0	0	0	0	
1	0	0	0	0	0	1	1	0	1	1	(#)

译码器的逻辑表达式

$N_3 = R_2C_0' + R_3C_1'$

$N_2 = R_1 + R_2C_0$

$N_1 = R_0C_0' + R_2'C_2 + R_1'R_0'C_0$

$N_0 = R_1C_1 + R_1'C_2 + R_3'R_1'C_1'$

4.11.4 控制器

图 4.45 给出了键盘扫描器的状态图。在状态 S_1，输出 $C_1=C_2=C_3=1$，键盘扫描器在此状态下等待直到有键被按下。在状态 S_2，C_0=1。因此如果按下的键在列 0 中，则有 K=1，且电路输出一个有效信号并进入到状态 S_5。由于已经对按键进行去抖动处理了，所以用信号 K 代替信号 Kd。如果列 0 中没有键被按下，则在状态 S_3，将对列 1 进行检测；如果有必要，在状态 S_4 对列 2 进行检测。在状态 S_5，所有的键均被释放后，电路复位，且 Kd 在复位前变为 0。

图 4.45 所示状态图在许多情况下都可以正常工作，但是它确实存在一定的时序问题。下面我们针对这些问题加以讨论。

1. 无论何时当一个键被按下时，K 永远为真吗？
   不是的。虽然当行信号 R_1, R_2, R_3 或 R_4 中有任何一个为真时 K 即为真，但是如果列扫描信号没有被激活，即使有键被按下，$R_1 \sim R_4$ 中的任何一个也不可能为真。
2. 当连续按下一个键时，Kd 会为假吗？
   是的。信号 Kd 其实就是把信号 K 延迟两个时钟周期。在扫描过程中，即使有键被按下，

信号 K 也可能为 0。例如，当键盘最右边一列中有一个键被按下，在扫描前两列时，K 为 0。不论何时 K 为 0，信号 Kd 在两个时钟周期后就为 0。因此，K 和 Kd 中任何一个都为真都不能说明有键被按下。

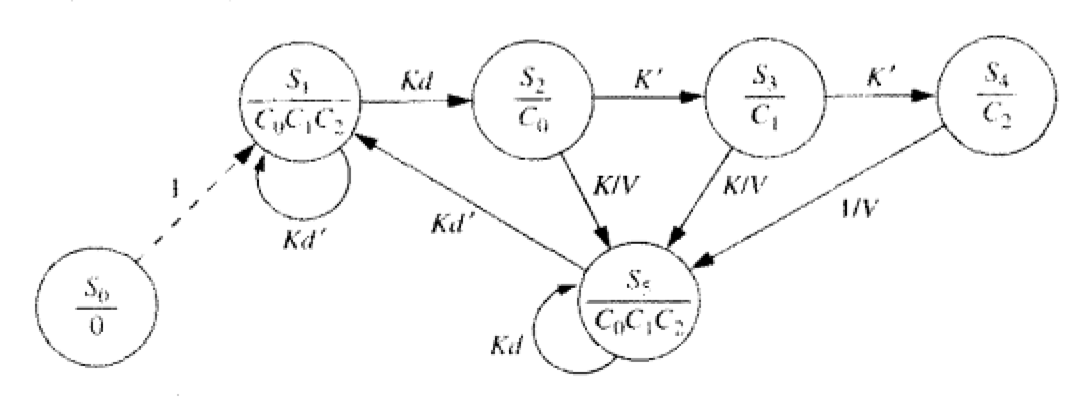


图 4.45　键盘扫描器的状态图

3. 当一个键被持续按下时，系统可以从状态 S_5 变为状态 S_3 吗？
   状态图见 4.45 所示。当 Kd 为假时会发生状态 S_4 到状态 S_5 的转变。在对 C_0 和 C_1 进行扫描时，Kd 有可能会为假。因此，当一个键被持续按下时，有可能电路会回到状态 S_1。例如，假设列 C_2 中有一个键被按下，我们将在状态 S_4 检测到此情况。但是，在对状态 S_2 和 S_3 的扫描过程中 K 为 0；因此，即使此键被持续按下，两个时钟周期后 Kd 也为 0。状态 S_4 中，我们可以找到正确的按键；但是如果 Kd 仍旧为 0，则系统会到达状态 S_5，系统就会出现故障。状态 S_5 意味着我们将放开被按下的键。但是 Kd 与按键动作不是同步的，那么 Kd' 就不能真实的表示按下的键是否被释放。由于在键被持续按下时，信号 Kd' 也会出现，所以如果在状态 S_5 时，由于之前状态中的扫描操作会使 Kd' 为真，所以即使没有释放按键，系统也可以从状态 S_5 变为状态 S_1。在这种情况下，同一个键可能被多次读取。
4. 如果一个键只被按下一个或两个时钟周期，那么会出现什么情况？
   如果一个键被快速的按下或释放，则系统就会出现问题，尤其是当按键在第三列时。当扫描器到达状态 S_4 时，按键可能已经被释放。因此，键应该被按下足够长的时间，这样扫描器才能走状态图中最长的一条路径，从 S_0 走到 S_5。这可能并不是什么大问题，因为数字系统的时钟一般都比机械开关操作的时间要短的多。

如果我们假设只有当 Kd 为真时，系统才能到达状态 S_5，那么这些问题就都可以得到解决。更改后的状态图见图 4.46。在转变为状态 S_5 之前，电路将在状态 S_2, S_3 和 S_4 等待，直到 Kd 变为 1，系统才变为状态 S_5。

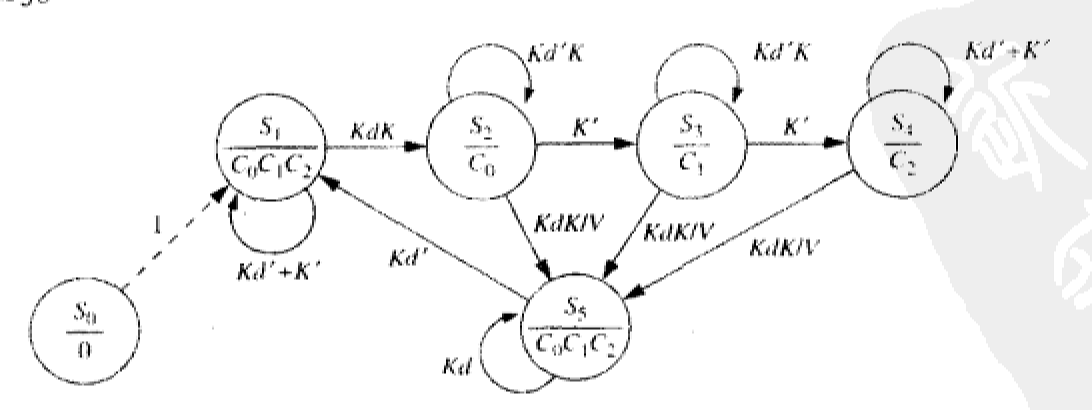


图 4.46　键盘扫描器的修改状态图

4.11.5　VHDL 代码

实现设计的 VHDL 代码示于图 4.47。译码器表达式、K 和 V 的表达式在程序中均使用并发语

句实现。进程语句用来实现键盘扫描和去抖动触发器的下一状态表达式。

```
entity scanner is
  port(R0, R1, R2, R3, CLK: in bit;
       C0, C1, C2: inout bit;
       N0, N1, N2, N3, V: out bit);
end scanner;
architecture behavior of scanner is
signal QA, K,Kd: bit;
signal state, nextstate: integer range 0 to 5;
begin
  K <= R0 or R1 or R2 or R3;   -- this is the decoder section
  N3 <= (R2 and not C0) or (R3 and not C1);
  N2 <= R1 or (R2 and C0);
  N1 <= (R0 and not C0) or (not R2 and C2) or (not R1 and not R0 and C0);
  N0 <= (R1 and C1) or (not R1 and C2) or (not R3 and not R1 and not C1);

  process(state, R0, R1, R2, R3, C0, C1, C2, K, Kd, QA)
  begin
    C0 <= '0'; C1 <= '0'; C2 <= '0'; V <= '0';
    case state is
      when 0 => nextstate < = 1;
      when 1 => C0 <= '1'; C1 <= '1'; C2 <= '1';
        if (Kd and K) = '1' then nextstate <= 2;
        else nextstate <= 1;
        end if;
      when 2 => C0 <= '1';
        if (Kd and K) = '1' then V <= '1'; nextstate <= 5;
        elsif K = '0' then nextstate <= 3;
        else nextstate <= 2;
        end if;
      when 3 => C1 <= '1';
        if (Kd and K) = '1' then V <= '1'; nextstate <= 5;
        elsif K = '0' then nextstate <= 4;
        else nextstate <= 3;
        end if;
      when 4 => C2 <= '1';
        if (Kd and K) = '1' then V <= '1'; nextstate <= 5;
        else nextstate <= 4;
        end if;
      when 5 => C0 <= '1'; C1 <= '1'; C2 <= '1';
        if Kd = '0' then nextstate <= 1;
        else nextstate <= 5;
        end if;
    end case;
  end process;

  process(CLK)
  begin
    if CLK = '1' and CLK'EVENT then
      state <= nextstate;
      QA <= K;
      Kd <= QA;
    end if;
  end process;
end behavior;
```

图 4.47　扫描器的 VHDL 代码

4.11.6　键盘扫描器的测试平台

仅靠观察比较输入 R_0, R_1, R_2 和 R_3 的波形来进行 VHDL 代码的检测是非常困难的，这是因为这些输入均取决于列输出（C_2, C_1 和 C_0）。一个较好的方法是使用一个用 VHDL 代码编写的测试平台来进行扫描器的测试。我们要测试的扫描器将被作为一个元件嵌入到测试平台程序中。测试平台内产生的信号与扫描器之间的接口见图 4.48 所示。测试平台靠提供适当的 R 信号模拟按键，响应从扫描器来的 C 信号。当测试平台从扫描器得到 V=1 时，它将检测 N 值所对应的键是否被按。

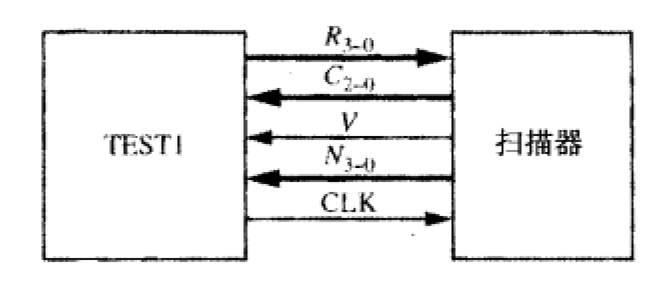


图 4.48 扫描器与测试平台的接口

```
library IEEE;
use IEEE.numeric_bit.all;

entity scantest is
end scantest;

architecture test1 of scantest is
component scanner
  port(R0, R1, R2, R3, CLK: in bit;
       C0, C1, C2: inout bit;
       N0, N1, N2, N3, V: out bit);
end component;

type arr is array (0 to 23) of integer;          -- array of keys to test
constant KARRAY: arr := (2,5,8,0,3,6,9,11,1,4,7,10,1,2,3,4,5,6,7,8,9,10,11,0);
signal C0, C1, C2, V, CLK, R0, R1, R2, R3: bit;   -- interface signals
signal N: unsigned(3 downto 0);
signal KN: integer;                            -- key number to test
begin
  CLK <= not CLK after 20 ns;                    -- generate clock signal

  -- this section emulates the keypad
  R0 <= '1' when (C0='1' and KN=1) or (C1='1' and KN=2) or (C2='1' and KN=3)
        else '0';
  R1 <= '1' when (C0='1' and KN=4) or (C1='1' and KN=5) or (C2='1' and KN=6)
        else '0';
  R2 <= '1' when (C0='1' and KN=7) or (C1='1' and KN=8) or (C2='1' and KN=9)
        else '0';
  R3 <= '1' when (C0='1' and KN=10) or (C1='1' and KN=0) or (C2='1' and KN=11)
        else '0';

  process                          -- this section tests scanner
  begin
    for i in 0 to 23 loop            -- test every number in key array
      KN <= KARRAY(i);              -- simulates keypress
      wait until (V = '1' and rising_edge(CLK));
      assert (to_integer(N) = KN)     -- check if output matches
        report "Numbers don't match"
        severity error;
      KN <= 15;                     -- equivalent to no key pressed
      wait until rising_edge(CLK);    -- wait for scanner to reset
      wait until rising_edge(CLK);
      wait until rising_edge(CLK);
    end loop;
    report "Test Complete.";
  end process;
  scanner1: scanner port map(R0,R1,R2,R3,CLK,C0,C1,C2,N(0),N(1),N(2),N(3),V);
                                 -- connect test1 to scanner
end test1;
```

图 4.49 扫描器测试平台的 VHDL 代码

键盘测试平台的 VHDL 代码见图 4.49 所示。扫描器被作为一个元件放于 *test*1 的结构体中，并通过端口映射与扫描器相连。用于检测的键号序列被存在 *KARRAY* 数组中。测试器使用并发语句对 R_0, R_1, R_2 和 R_3 进行模拟，进而模拟键区的运行。无论何时，只要 C_2, C_1, C_0 或键号(*KN*)发生变化，则各个 *R* 的新值将被计算出来。例如，如果 $KN = 5$（模拟按键 5），则 $R_0R_1R_2R_3$=0100 在 $C_0\,C_1C_2$=010 时被传送到扫描器中。测试过程如下：

1. 从数组中读键号以模拟一个按键。
2. 直到 V=1 并且时钟上升沿到来时，才开始工作。
3. 比较来自扫描器的输出 *N* 是否与键号相符。
4. 置 *KN*=15 来模拟没有键被按（因为 15 不是一个有效建号，所有的 *R* 都会变为 0）。
5. *Kd*=0 后在选择新键。

利用 *KARRAY* 中的不同数据，就可以尝试各种不同行号和列号的按键情况。测试平台使用 **assert** 语句来测试来自扫描器的输出是否与键号相符。同时，如果扫描器生成任何错误的键号，则 **report** 语句就会报错，而且当所有的键均被测试后，显示“测试完毕”。

4.12 二进制除法器的设计

4.12.1 无符号数除法器

下面我们考虑一个二进制正数的并行除法器的设计。首先我们设计一个电路用于计算 8 位数除以 4 位数，得到一个 4 位的商。下面的例子说明了除法的运算过程：

```
                              1010      商
除数             1101 / 10000111      被除数
                          1101
(135 ÷ 13 = 10 with        0111
         余数为 5)          0000
                             1111
                             1101
                              0101
                              0000
                              0101      余数
```

二进制乘法可以分解成一系列的相加和移位运算，除法也可以分解成一系列的减法和移位运算。我们将使用一个 9 位的被除数寄存器和一个 4 位的除数寄存器来构建除法器，如图 4.50 所示。在除法过程中，在每次相减前将被除数左移，而不是将输出右移。注意，我们需要在被除数寄存器左端增加一位，以避免被除数左移时丢失一位。不必使用一个单独的寄存器来存放商，我们可以随着被除数的左移，将商一位一位地放入被除数寄存器的最右边。

下面重新计算先前的除法例子(135 ÷ 13)。下面显示每个时钟周期内各位数字在寄存器中的位置。最初，被除数和除数输入如下：

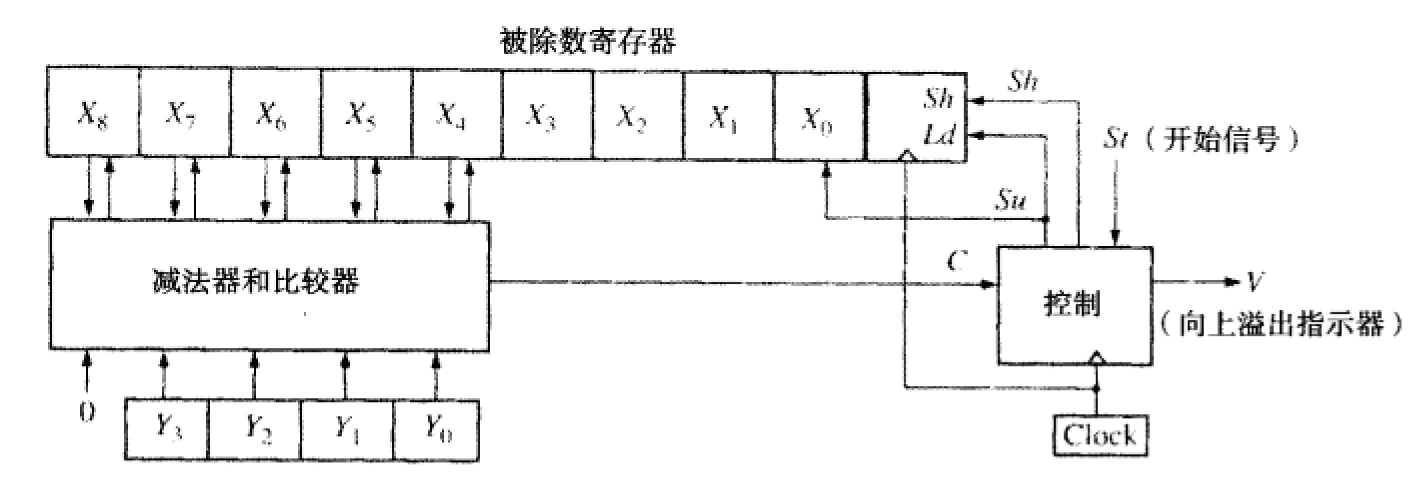
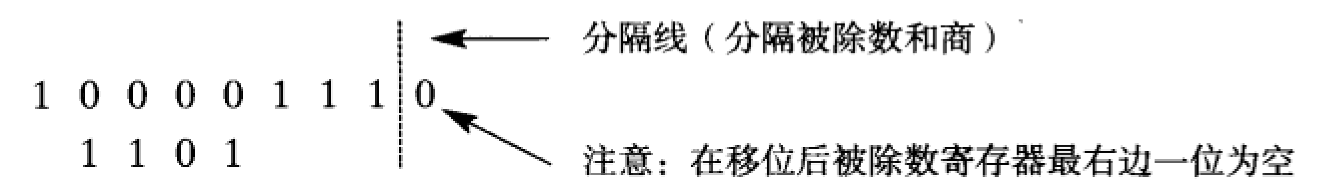


图 4.50 并行除法器实现框图

0	1	0	0	0	0	1	1	1
	1	1	0	1				

如果结果不为负，则不可以进行减操作，所以在相减之前，必须进行移位。不是将除数右移一位，而是将被除数左移一位：



现在可以做减法操作了，商的第一个位 1 被存放进被除数寄存器中未使用的位置：

0 0 0 1 1 1 1 1 ¦ 1 ← 商的第一位

接下来，将被除数左移一位：

0 0 1 1 1 1 1 ¦ 1 0
1 1 0 1

因为相减会得出一个负数结果，所以我们再次将被除数左移，商的第二位仍为 0：

0 1 1 1 1 1 ¦ 1 0 0
1 1 0 1

现在执行减法操作，将商的第三位 1 存入被除数寄存器未使用的位置：

0 0 0 1 0 1 ¦ 1 0 1 ← 商的第三位

进行最后一次移位，商的第四位为 0：

0 0 1 0 1 ¦ 1 0 1 0
余数 ¦ 商

最终的结果与第一个例子得到的结果相符。

如果除法运算所得结果（商）包含的位数比用来存储商的位数还要多，那么我们就可以说产生了向上溢出(overflow)。对于图 4.50 中的除法器，由于我们用 4 位来存放商，所以如果商大于 15，就可能产生向上溢出。实际上，不需要执行除法运算来判断是否存在溢出，因为初始时通过被除数和除数的比较就可以判断商是否会太大。例如，如果我们计算 135 除以 7，寄存器的初始

内容为

$$0\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1$$
$$0\ 1\ 1\ 1$$

由于减法的结果可以有非负的结果，所以我们应该从被除数中减去除数，然后把商的一位 1 放置在被除数寄存器的最右边。然而，我们却不能这么做，因为被除数寄存器的最右边存储着被除数的最低有效位，在这里输入一位商会破坏被除数。因此，对于我们分配的 4 位的空间，商显得太大而无法装入，所以我们检测到向上溢出。通常来说，对于图 4.50，如果一开始 $X_8X_7X_6X_5X_4 \geqslant Y_3Y_2Y_1Y_0$（即如果被除数寄存器的左边 5 位大于或等于除数），则商就会大于 15，就会产生向上溢出。注意，如果 $X_8X_7X_6X_5X_4 \geqslant Y_3Y_2Y_1Y_0$，则商为

$$\frac{X_8X_7X_6X_5X_4X_3X_2X_1X_0}{Y_3Y_2Y_1Y_0} \geqslant \frac{X_8X_7X_6X_5X_40000}{Y_3Y_2Y_1Y_0} = \frac{X_8X_7X_6X_5X_4 \times 16}{Y_3Y_2Y_1Y_0} \geqslant 16$$

除法器的运行可以根据图 4.50 中的方框图来解释。移位信号(Sh)会在下一时钟上升沿使被除数向左移一位。减信号(Su)使被除数寄存器的最左边 5 位减掉除数，并且将商位（被除数最右边一位）置 1。如果除数大于被除数的最左边的 5 位，则比较器的输出 C=0；否则 C=1。当 C=0 的时候，如果结果非负，则减法无法运行，所以产生一个移位信号。当 C=1 时，产生一个相减信号，商位置 1。控制电路生成移位信号和相减信号所需的序列。

图 4.51 给出了控制电路的状态图。当开始信号(St)出现，8 位被除数和 4 位除数被载入相应的寄存器。如果 C 为 1，则商需要 5 位或者更多的位。因为提供的存储空间只够存放一个 4 位的商，这种情况会产生向上溢出，所以除法运算将被停止，而输出 V 使溢出指示器置位。通常情况下，C 的初始值为 0，所以首先进行移位操作，控制电路进入状态 S_2。然后如果 C=1，进行相减操作。相减完成后，C 总是等于 0，所以在下一个时钟会产生移位操作。这个过程将一直持续，直到进行了 4 次移位，此时控制电路处于状态 S_5。然后，如果需要则进行最后一次相减操作。最后控制电路回到停止状态。对于这个例子，我们假设，当开始信号(St)出现时，它将会在一个时钟内保持为 1，然后变为 0，并持续到控制电路回到状态 S_0 为止。因此，St 从 S_1 到 S_5 直接总是等于 0。

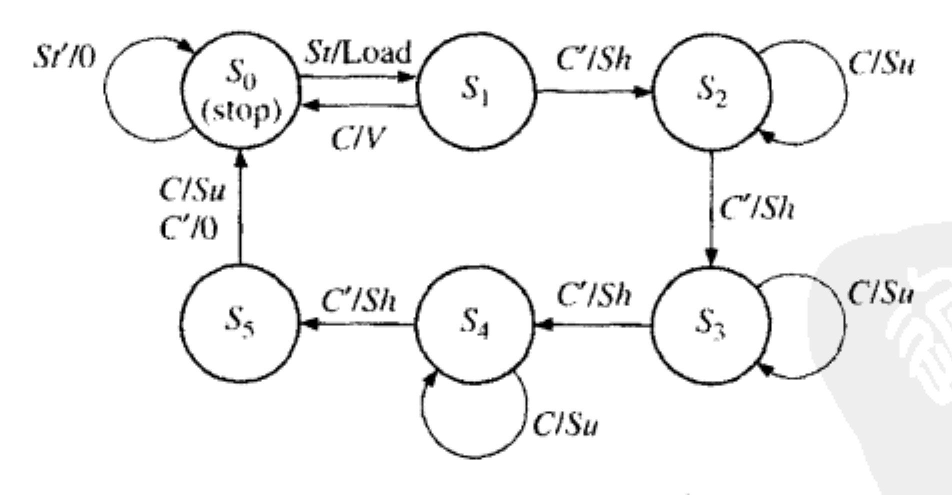


图 4.51 除法器控制电路的状态图

表 4.5 给出了控制电路的状态表。因为我们假设在状态 S_1, S_2, S_3 和 S_4 中，St 均为 0，所以当 St=1 时，这几个状态的下一个状态和输出均为“随意项”。输出表中的各项表示何时输出为 1。例如，表中出现 Sh 时，表示 Sh=1 且其他输出为 0。

表 4.5 除法器控制电路的状态表

State	*StC* 00	01	11	10	*StC* 00	01	11	10
S_0	S_0	S_0	S_1	S_1	0	0	Load	Load
S_1	S_2	S_0	—	—	*Sh*	V	—	—
S_2	S_3	S_2	—	—	*Sh*	*Su*	—	—
S_3	S_4	S_3	—	—	*Sh*	*Su*	—	—
S_4	S_5	S_4	—	—	*Sh*	*Su*	—	—
S_5	S_0	S_0	—	—	0	*Su*	—	—

这个例子给出了无符号二进制数除法器设计的一般方法，而且此设计可以很容易地扩展到更大位数的除法器。例如，16 位除以 8 位的除法器，32 位除 16 位的除法器等。下面我们将设计一个有符号二进制数（二进制补码）除法器：被除数为 32 位，除数为 16 位，输出的商为 16 位。

4.12.2 有符号数除法器

现在，我们为有符号（二进制补码）二进制数设计一个除法器，其中被除数为 32 位，除数为 16 位，商为 16 位。尽管有直接对有符号数作除法的算法，但是这种算法过于复杂，所以我们使用简单的方法。如果被除数和除数为负，则对它们取补；当完成除法时，如果商为负，则对其取补。

图 4.52 给出了此除法器的框图。我们使用一个 16 位总线为寄存器载入数据。因为被除数是 32 位的，所以需要两个时钟分别把被除数寄存器的高位部分和低位部分载入，再需要一个时钟来载入除数。此外，我们还需要一个符号触发器来存储被除数的符号。我们使用的被除数寄存器中内嵌取补器。减法器由一个加法器和一个取补器组成，所以只要将除数的二进制补码加到被除数寄存器，就完成了减操作。如果除数为负，则无需单独进行一次取补操作，我们只需使取补器不起作用，直接加上负的除数，而不是加上除数的补码。控制电路分为两部分：主控部分（决定移位和相减的操作顺序）和计数器（计算移位的次数）。当进行 15 次移位后，计数器输出一个信号 K=1。控制信号定义如下：

LdU 从总线上读取被除数的高位部分
LdL 从总线上读取被除数的低位部分
Lds 把被除数的符号载入到符号寄存器中
S 被除数的符号
Cm1 对被除数寄存器取补（二进制补码）
Ldd 从总线上读入除数
Su 加法器输出到总线(*Ena*)并从总线上读取被除数的高位部分
Cm2 使取补器开始工作（*Cm2* 等于除数符号位取反，所以当除数为正时，要对其取补；若除数为负，则无需取补）
Sh 使被除数寄存器左移移位，并对计数器加 1
C 加法器进位输出（如果 $C = 1$，则从被除数中减去除数）
St 开始信号
V 溢出
Qneg 商为负（当被除数的符号和除数的符号不同时，$Qneg = 1$）

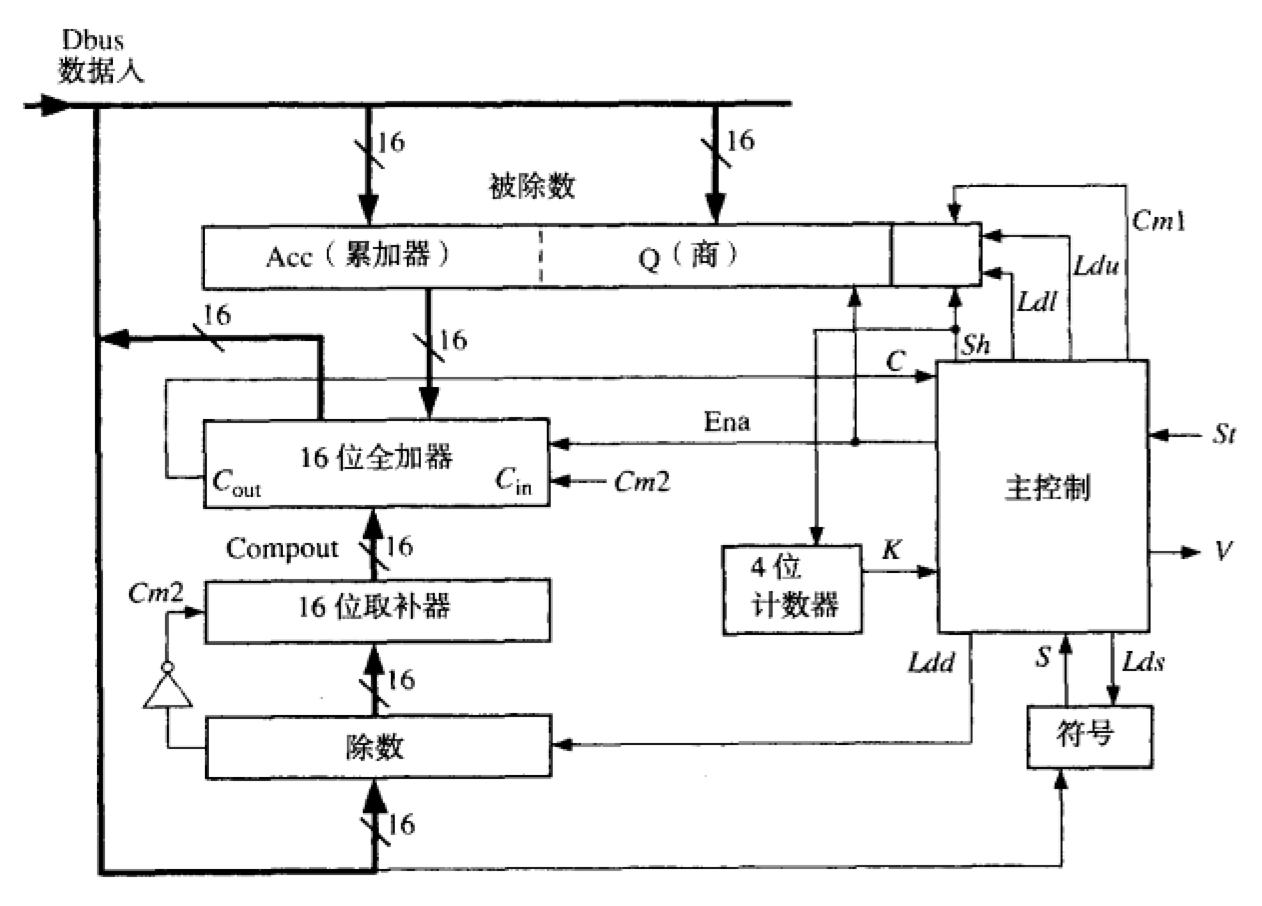


图 4.52 有符号数除法器的框图

有符号数除法的操作步骤如下：

1. 从总线载入被除数的高位部分，将被除数的符号复制到符号触发器。
2. 从总线载入被除数的低位部分。
3. 从总线载入除数。
4. 如果被除数为负，则对其求补。
5. 如果出现向上溢出，则进入完成状态。
6. 否则通过一系列的移位和相减完成除法操作。
7. 当除法完成，如果需要，则对商求补，进入完成状态。

有符号数除法对向上溢出的检测比无符号数除法溢出检测要稍微复杂一些。首先，要考虑所有的正数情况。因为除数和商都是 15 位再加上符号位，所以最大值为 7FFFh。因为余数必须比除数小，则余数最大值为 7FFEh。因此，不产生向上溢出的被除数最大为

$$除数\times商 + 余数 = 7FFFh \times 7FFFh + 7FFEh = 3FFF7FFFh$$

如果被除数比 1(3FFF8000h)大，则除以 7FFFh（或者任何比它小的数），就会产生向上溢出。如果要测试是否存在向上溢出，则需要将被除数左移一位，然后对除数和被除数的高位部分(*divu*)进行比较，如果 *divu*≥除数，商就会大于最大值，就会产生向上溢出。对于前面的例子，将 3FFF8000h 左移一位得到 7FFF0000h。因为 7FFFh 等于除数，所以产生向上溢出。另一方面，将 3FFF7FFFh 左移一位得到 7FFEFFFEh，因为 7FFE 小于 7FFF，所以除以 7FFF 不会产生向上溢出。

在溢出检测前，我们就要对被除数进行左移，下面介绍另外一种溢出检验方法：如果我们将被除数左移，得到 *divu*≥除数，则进行减操作，生成一个商位 1。然后，这一位就必定会进入商的符号位。这就会使商为负，这是不正确的。在溢出测试完毕后，我们必须再次对被除数移位，并在符号位之后提供一个位置用以存储商的第一位。由于在进行除法运算时，我们对负的被除数和除数都取补后再进行操作，所以除了被除数等于 80000000h（最大的负数）这种特殊的情况外，

此种检测溢出的方法对于负数同样适用。对于这种特殊情况的溢出检测，将在习题中解决。

图 4.53 给出了控制电路的状态图。当 St=1，寄存器载入数据。在状态 S_2，如果被除数的符号(S)为 1，对被除数取补。在状态 S_3，将被除数左移一位，然后在状态 S_4 进行溢出测试。如果 C=1，则可以进行减操作，如果有溢出，则电路进入完成状态。否则，被除数左移。在状态 S_5，对 C 进行测试，如果 C=1，则 Su=1，意味着 Ldu 和 Ena 均有效，所以在总线上加法器的输出有效，并被载入到被除数寄存器的高位部分完成减操作。否则，Sh=1，并把被除数寄存器移位。操作继续，直到 K=1。如果 C=0，则最后进行一次移位，电路进入 S_6。然后如果除数的符号与存储的被除数的符号不同，则对被除数寄存器取补，这样商就会得到正确的符号。

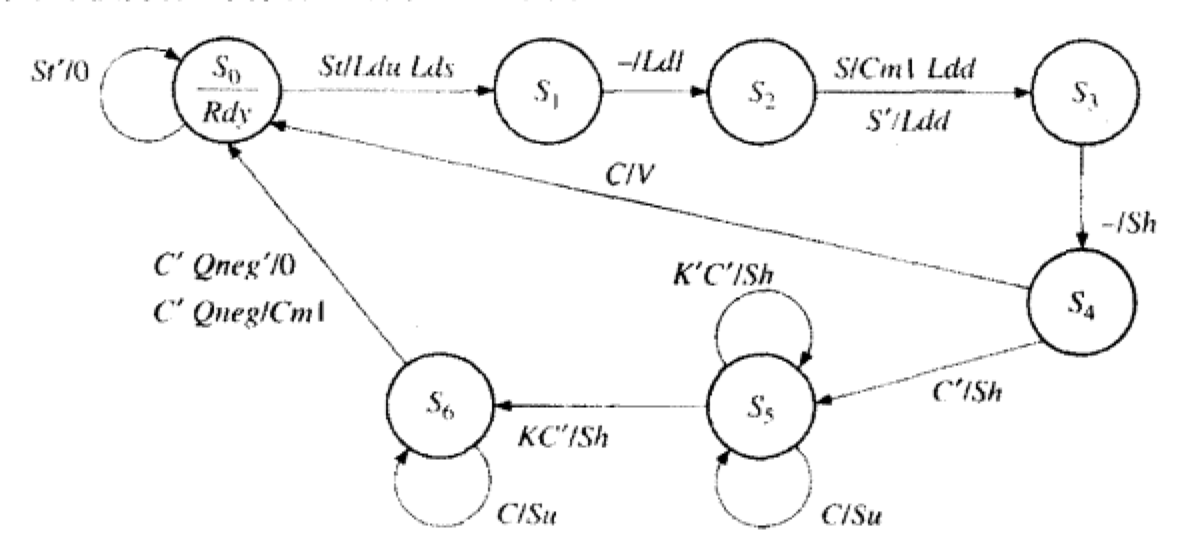


图 4.53 有符号除法器控制电路的状态图

有符号数除法器的 VHDL 代码如图 4.54 所示。因为二进制反码生成器和加法器均为组合电路，所以使用并发语句对其进行表述。所有表示寄存器输出的信号都在时钟上升沿进行数值更新，所以这些信号在 CLK 变为 '1' 后，才在进程中被更新。计数器用一个信号 $count$ 进行模拟。为了方便地列出仿真器的输出，我们加入了一个准备信号(Rdy)。此信号在状态 S_0 时启动，表述除法操作的完成。

```
library IEEE;
use IEEE.numeric_bit.all;

entity sdiv is
  port(CLK, St: in bit;
       Dbus: in unsigned(15 downto 0);
       Quotient: out unsigned(15 downto 0);
       V, Rdy: out bit);
end sdiv;

architecture Signdiv of Sdiv is
signal State: integer range 0 to 6;
signal Count: unsigned(3 downto 0); -- integer range 0 to 15
signal Sign, C, Cm2: bit;
signal Divisor, Sum, Compout: unsigned(15 downto 0);
signal Dividend: unsigned(31 downto 0);
alias Acc: unsigned(15 downto 0) is Dividend(31 downto 16);
begin                                         -- concurrent statements
  Cm2 <= not divisor(15);
  compout <= divisor when Cm2 = '0'           -- 1's complementer
             else not divisor;
  Sum <= Acc + compout + unsigned'(0=>Cm2);  -- adder output
  C <= not Sum(15);
  Quotient <= Dividend(15 downto 0);
  Rdy <= '1' when State = 0 else '0';
  process(CLK)
```

图 4.54 32 位有符号除法器的 VHDL 模型

```
  begin
    if CLK'event and CLK = '1' then  -- wait for rising edge of clock
      case State is
        when 0 =>
          if St = '1' then
            Acc <= Dbus;                           -- load upper dividend
            Sign <= Dbus(15);
            State <= 1;
            V <= '0';                              -- initialize overflow
            Count <= "0000";                       -- initialize counter
          end if;
        when 1 =>
          Dividend (15 downto 0) <= Dbus;            -- load lower dividend
          State <= 2;
        when 2 =>
          Divisor <= Dbus;
          if Sign = '1' then  -- two's complement Dividend if necessary
            dividend <= not dividend + 1;
          end if;
          State <= 3;
        when 3 =>
          Dividend <= Dividend(30 downto 0) & '0';     -- left shift
          Count <= Count+1; State <= 4;
        when 4 =>
          if C = '1' then                              -- C
            v <= '1'; State <= 0;
          else                                         -- C'
            Dividend <= Dividend(30 downto 0) & '0';   -- left shift
            Count <= Count+1; State <= 5;
          end if;
        when 5 =>
          if C = '1' then                              -- C
            ACC <= Sum;                                -- subtract
            dividend(0) <= '1';
          else
            Dividend <= Dividend(30 downto 0) & '0';   -- left shift
            if Count = 15 then State <= 6; end if;     -- KC'
            Count <= Count+1;
          end if;
        when 6 =>
          state <= 0;
          if C = '1' then                              -- C
            Acc <= Sum;                                -- subtract
            dividend(0) <= '1'; State <= 6;
          elsif (Sign xor Divisor(15)) = '1' then      -- C'Qneg
            Dividend <= not Dividend + 1;
          end if;                                      -- 2's complement Dividend
      end case;
    end if;
  end process;
end signdiv;
```

图 4.54（续） 32 位有符号除法器的 VHDL 模型

现在我们准备用 VHDL 仿真器对除法器进行测试。我们需要针对除法过程中会出现的各种不同的特殊情况，对程序进行综合测试。首先，要对除数和被除数的不同符号组合(++, +–, –+, —)进行基本操作的测试。也需要在这四种情况下进行溢出情况的测试，还要对受限情况进行测试，包括最大商、0 商等。因为测试数据要在一定的时间按照一定的顺序输入，而且完成除法所需的时间长度取决于测试数据，所以我们使用了 VHDL 测试平台。图 4.55 给出了除法器的测试平台。此测试平台含有测试数据的被除数数组和除数数组。符号 X “07FF00BB” 为数据串的 16 进制表示。在 testsdiv 中，进程首先将被除数的高位部分放入 *Dbus*，并产生开始信号。时钟到来后，将被除数的低位部分放置到 *Dbus* 上，然后系统等待 *Rdy* 信号，接收到此信号表示除法操作完成。*Count* 的值与循环变量相同，因此可以用 *Count* 的改变触发列表中的输出。

```
library IEEE;
use IEEE.numeric_bit.all;

entity testsdiv is
end testsdiv;

architecture test1 of testsdiv is
component sdiv
  port(CLK, St: in bit;
       Dbus: in unsigned(15 downto 0);
       Quotient: out unsigned(15 downto 0);
       V, Rdy: out bit);
end component;

constant N: integer : = 12;                          -- test sdiv1 N times
type arr1 is array(1 to N) of unsigned(31 downto 0);
type arr2 is array(1 to N) of unsigned(15 downto 0);
constant dividendarr: arr1 := (X"0000006F", X"07FF00BB", X"FFFFFE08",
     X"FF80030A", X"3FFF8000", X"3FFF7FFF", X"C0008000", X"C0008000",
     X"C0008001", X"00000000", X"FFFFFFFF", X"FFFFFFFF");
constant divisorarr: arr2 := (X"0007", X"E005", X"001E", X"EFFA", X"7FFF",
   X"7FFF", X"7FFF", X"8000", X"7FFF", X"0001", X"7FFF", X"0000");
signal CLK, St, V, Rdy: bit;
signal Dbus, Quotient, divisor: unsigned(15 downto 0);
signal Dividend: unsigned(31 downto 0);
signal Count: integer range 0 to N;

begin
  CLK <= not CLK after 10 ns;
  process
  begin
    for i in 1 to N loop
      St <= '1';
      Dbus <= dividendarr(i) (31 downto 16);
      wait until (CLK'event and CLK = '1');
      Dbus <= dividendarr(i) (15 downto 0);
      wait until (CLK'event and CLK = '1');
      Dbus <= divisorarr(i);
      St <= '0';
      dividend <= dividendarr(i) (31 downto 0);   -- save dividend for listing
      divisor <= divisorarr(i);                   -- save divisor for listing
      wait until (Rdy = '1');
      count <= i;                                  -- save index for triggering
    end loop;
  end process;
  sdiv1: sdiv port map(CLK, St, Dbus, Quotient, V, Rdy);
end test1;
```

图 4.55　有符号数除法器的测试平台

图 4.56 给出了仿真器的命令文件和输出。由于所列的语句中使用了`-NOtrigger` 和 `-Trigger`，所以只有当 *count* 信号发生改变时，才会显示输出结果。仿真器输出的测试结果表明，除了下面的特殊情况外，此除法器在其他测试情况下都是正确的：

$$\text{C0008000h} \div \text{7FFFh} = -\text{3FFF8000} \div \text{7FFFh} = -\text{8000h} = \text{8000h}$$

在这种情况下，只要除法计算产生了向上溢出，除法就无法正确完成。一般来说，只要商为 8000h(最小负数)，除法器就会指示存在向上溢出。因为基本上除法器都是除以正数，而最大的正商为 7FFFh，所以会发生这种情况。如果生成的商必须能够为 8000h，那么就要对溢出检测进行修改，这样就不会在这种特殊情况下产生向上溢出了。

在这章中，我们介绍了很多设计实例，有算术电路，也有非算术电路。七段数字显示器、BCD 加法器、交通灯控制器、记分板和键盘扫描器均为非算术实例。我们也介绍了一些算术实例，如无符号数和有符号数的加法、乘法和除法等。同时，我们还对一些特殊设计进行讨论，如提前进位加法器和阵列乘法器等。我们使用算法实现了这些数字系统。对这样的系统设计了框图，定义

了所需的控制信号，并使用状态图来定义合适的操作顺序，生成控制信号的时序，并利用 VHDL 在不同的层面描述系统。因此，我们可以对所设计的系统操作进行正确的仿真和测试。

```
-- Command file to test results of signed divider
add list -hex -NOtrigger dividend divisor Quotient V -Trigger count
run 5300

  ns   delta   dividend   divisor   quotient   v   count
   0     +0    00000000     0000       0000    0      0
 470     +3    0000006F     0007       000F    0      1
 910     +3    07FF00BB     E005       BFFE    0      2
1330     +3    FFFFFE08     001E       FFF0    0      3
1910     +3    FF80030A     EFFA       07FC    0      4
2010     +3    3FFF8000     7FFF       0000    1      5
2710     +3    3FFF7FFF     7FFF       7FFF    0      6
2810     +3    C0008000     7FFF       0000    1      7
3510     +3    C0008000     8000       7FFF    0      8
4210     +3    C0008001     7FFF       8001    0      9
4610     +3    00000000     0001       0000    0      A
5010     +3    FFFFFFFF     7FFF       0000    0      B
5110     +3    FFFFFFFF     0000       0002    1      C
```

图 4.56 有符号除法器的仿真测试结果

习题

4.1 设计一个 BCD 加法器的校正电路，该加法器在状态 S_0 时对 Z digit0 和 C 进行计算（见图 4.5 和 4.6）。当 $S_0>9$ 时，校正电路把“0110”加到 S_0 上，这就等价于在 S_0 上加“0AA0”，且当 $S_0>9$ 时 A='1'。 画出校正电路的框图，此电路由一个全加器，三个半加器和一个用于计算 A 的逻辑电路构成。在为 A 设计电路时，要求使用的门电路数目最小。

注意：S_0 的最大可能值为 10010。

4.2 (a) 如果门延迟为 5 ns，则最快的 4 位行波进位加法器的延迟是多少？并做出解释。

(b) 如果门延迟为 5 ns，则最快的 4 位加法器的延迟是多少？是何种加法器？为什么？

4.3 使用图 4.10 中给出的 4 位加法器作为一个元件，写出 16 位先行进位加法器的 VHDL 模块。

4.4 当计算 0101 1010 1111 1000 与 0011 1100 1100 0011 的和时，若使用图 4.9 中的 16 位先行进位加法器，请推导其生成信号、传递信号、组生成信号、组传递信号、最后结果（和）及进位。

4.5 (a) 写出带有累加器的一位全加器的 VHDL 代码。该模块有两个控制输入 Ad 和 L。当 Ad=1 时，输入 Y (和进位输入)加到累加器中。如果 L=1，则输入 Y 被载入累加器。

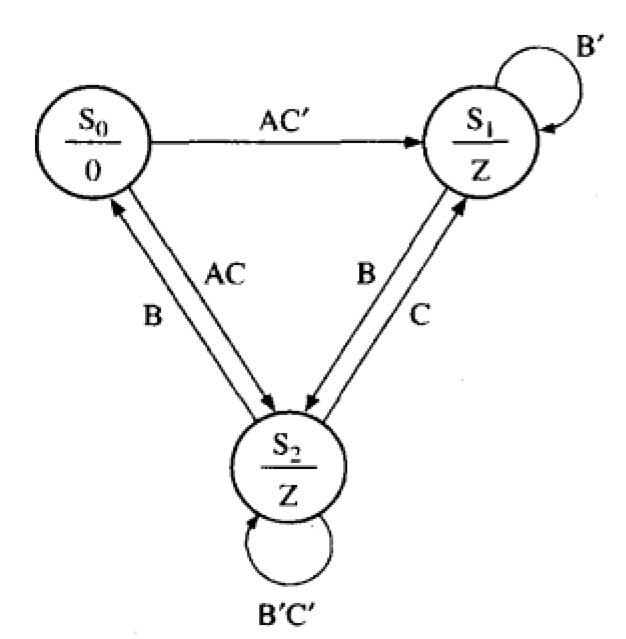
(b) 编写 4 位全减器 VHDL 代码，可以使用(a)中定义的模块。假设负数用反码表示，且全减器的控制输入为 Su(减)和 Ld（载入数据）。

4.6 (a) 使用一个具有加法逻辑的模 13 计数器实现如图 4.14 所示的交通灯控制电路，在每个时钟周期，计算器都向上计数（有两种情况除外）。使用一个 ROM 生成输出。

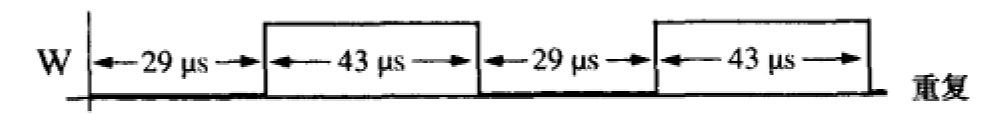
(b) 写出(a)中设计的 VHDL 程序。

(c) 为(b)编写一个测试平台，并验证你所设计的控制器是否能正常工作，要求使用并发语句生成 Sa 和 Sb 的测试输入。

4.7 对下面的状态图增加一些必要的部分使之成为一个完整的状态图，并对答案的正确性进行说明，把此状态图转化为状态表，用 0 和 1 描述输入和输出。



4.8 写出可以生成如下波形(*W*)的可综合的 VHDL 代码。要求只使用一个进程并假设可以使用一个周期为 1 μs 的输入时钟。



4.9 一个 BCD 加法器可以求两个 BCD 数的和（每个数的取值范围均为 0 ~ 9），而且求得的和也是 BCD 格式的。例如，如果求 9（1001）和 8（1000）的和，则结果为 17（10111）。请使用一个 4 位二进制加法器和适当的控制电路实现此加法器。假设这两个 BCD 数已经分别载入到两个 4 位寄存器（*A* 和 *B*）中。另外提供一个 5 位的和寄存器。你需要对求得的结果做一定的校正，使其为 BCD 格式，因为二进制加法器产生的结果的取值范围为 0000 ~ 1111（有些情况下还要加上进位）。如果校正时需要进行额外的加操作，请使用同一个加法器（也就是说你只能使用一个加法器）。在加法器输入端使用多路选择器选择恰当的数字在每次循环中进行加操作。假设用一个开始信号对加法器进行初始化和一个结束信号指示操作的完成。

(a) 画出系统框图，正确地标注出每个元件，要求标注出其功能和大小。

(b) 说明实现此加法操作的算法的详细步骤，并对校正步骤进行解释和说明。

(c) 画出控制电路的状态图。

4.10 写出由 1 个 16 位移位寄存器，1 个控制器和 1 个 4 位向下计数器构成的移位寄存器模块的 VHDL 代码。移位寄存器可以移动位数的多少取决于模块中计数器的大小。模块的输入为数 *N*（表示移位数且取值范围为 1 ~ 15）、16 位矢量 *par_in*、时钟信号和开始信号 *St*。当 *St*=‘1’时，*N* 被载入到向下计数器中，接着移位寄存器循环左移 *N* 次，控制器回到开始状态。假设 *St* 为‘1’，只持续一个时钟周期且所有操作均在时钟下降沿进行同步。

(a) 画出系统框图，并定义所需的控制信号。

(b) 画出控制器的状态图（2 个状态）。

(c) 写出此移位寄存器模块的 VHDL 代码。要求使用两个进程（一个用于电路的组合部分，一个用于更新寄存器）。

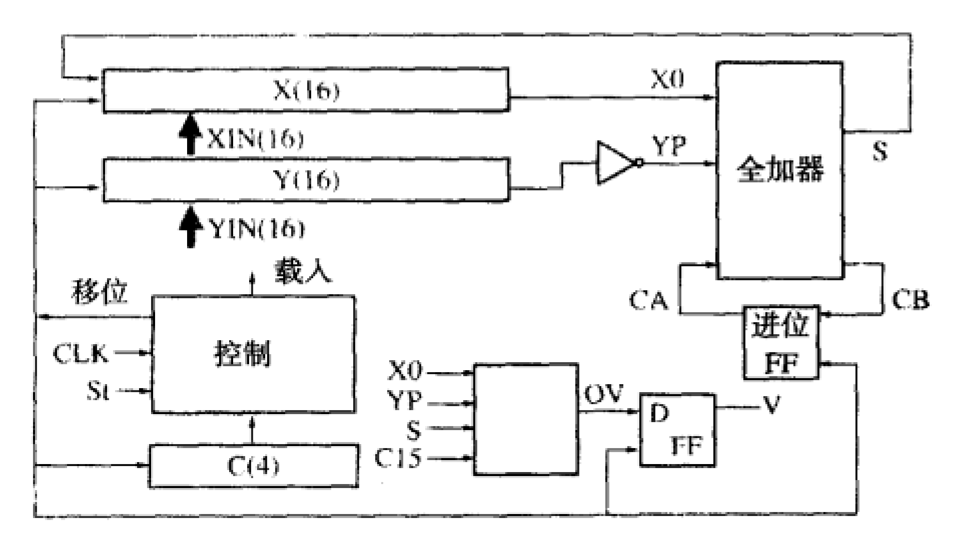
4.11 (a) 图 4.12 给出了一个带有累加器的 32 位串行加法器的框图。控制电路使用一个 5 位计数器，当其处于状态 11111 时，输出信号 *K*=1。当接收到开始信号(*N*)后，寄存器载入数据。假设在加运算结束前，*N* 均为 1。当加运算结束时，控制电路变为停止状态，并且一直保持该状态直到 *N* 变回 0。画出控制电路的状态图（计数器不计入）。

(b) 写出整个系统的 VHDL 代码，并验证其操作的正确性。

4.12 一个 16 位二进制补码串行减法器框图如下图所示。当 *St*=1 时，寄存器载入数据，然后开始进行减法运算。移位计数器 *C* 在移位 15 次后输出信号 *C*15=1。当存在向上溢出时，*V* 置 1。在置数时进位触发器置 1，以保证补码的形成。假设 *St*=1 持续一个时钟时间。

(a) 画出控制器的状态图（2 状态）。

(b) 写出此系统的 VHDL 代码。使用两个进程，第一个进程定义下一状态和控制信号；第二个进程在时钟上升沿更新各个寄存器。



4.13 设计一个 BCD 码–二进制码转换器。初始化时，寄存器 *A* 中有一个 3 数字 BCD 码。当接收到信号 *St* 时，开始把此 BCD 码转换为二进制码，并把转换后得到的二进制数存储在寄存器 *B* 中。转换的每一步均把整个 BCD 码（连同二进制数）右移一位。如果在某个数字位上所得的结果超过或等于 1000，则校正电路便从结果中减去 0011（如果结果小于 1000，则校正电路不对结果作任何改变）。使用移位计数器记录移位的个数。转换结束后，*B* 的最大值为 999（二进制）。注意：*B* 为 10 位。

(a) 画出此 BCD–二进制数转换器的框图。

(b) 画出此 BCD–二进制数转换器的框图。

(c) 画出控制电路的状态图（3 状态）。使用下列控制信号：

St 开始转换；*Sh* 右移；*Co* 减操作校正；*C*9：计数器在第 9 状态；*C*10 计数器在第 10 状态（可以使用 *C*9,*C*10 中的任意一个，但是不能同时使用）。

(d) 写出此系统的 VHDL 程序。

4.14 设计用减奇整数的方法求一个 8 位无符号二进制数 *N* 的平方根的电路。为了得到 *N* 的平方根，我们把它减去 1, 3, 5，等等，直到再减去任何一个奇整数原数就会是负数为止。例如计算 $\sqrt{27}$：27–1=26；26–3=23；23–5=18；18–7=11；11–9=2；2–11（不能再减了）。我们进行了 5 次减运算。所以 $\sqrt{27}$ =5。注意最后一个奇整数为 $11_{10}=1011_2$，此奇整数中包含平方根，去掉其二进制表示的最右边的 1，即可得到平方根 ($101_2=5_{10}$)。

(a) 画出平方根器的框图，包含一个存储数 *N* 的寄存器、一个减法器、一个存储各个奇整数的寄存器和一个控制电路。指出在何处读出平方根的值，对图中的控制信号加以定义。

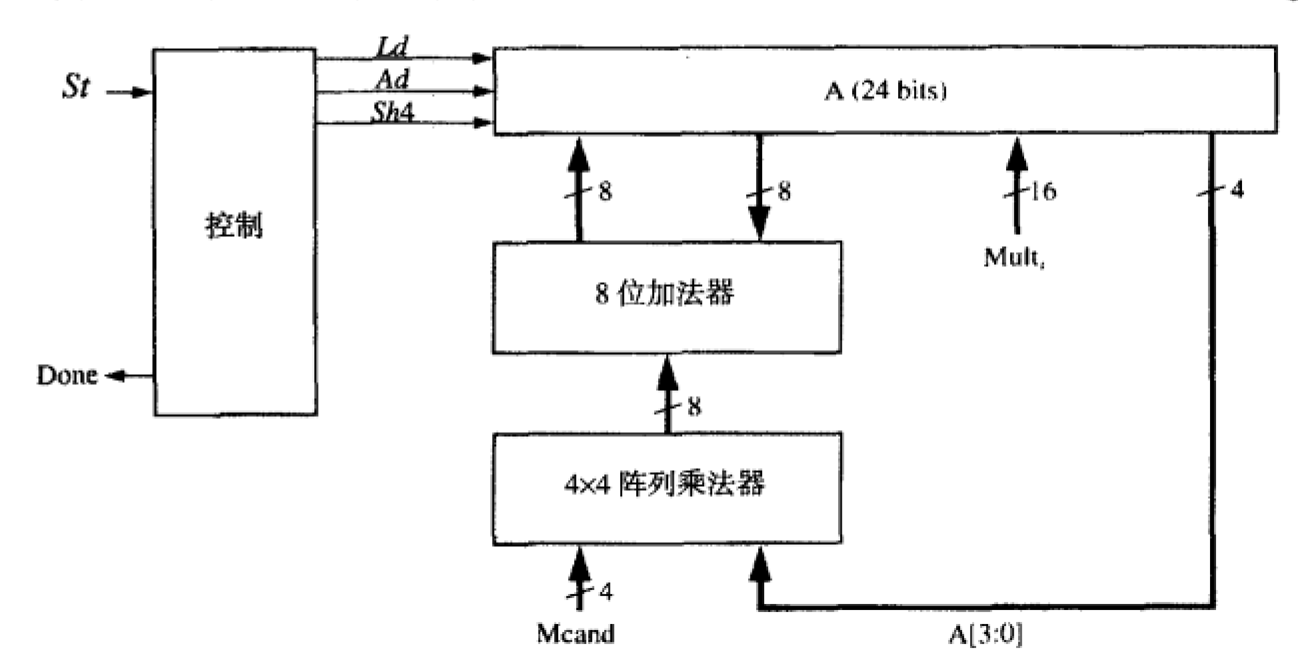
(b) 使用最少的状态数实现控制电路，并画出控制电路的状态图。存储数 *N* 的寄存器在 *ST*=1 时载入数据。当平方根计算完毕，控制电路应输出一个完成信号，并等到 *ST*=0 时再重新启动。

4.15 一个无符号二进制数乘法器可以求一个 4 位数和一个 16 位数的积，而且积为 20 位，为了

加快此乘法器的速度，我们使用一个 4×4 阵列结构乘法器，这样我们就可以在一个时钟内对 4 位作乘法，而不是 1 位。硬件使用一个 24 位累加寄存器，通过使用一个控制信号 *Sh4*，此寄存器可以每次右移 4 位。阵列结构乘法器可以每次计算两个 4 位数的积，且积为 8 位。通过使用控制信号 *Ad*，数组乘法器的结果加到累加器中。当 *St* 信号出现时，16 位乘数载入到寄存器 *A* 的低位部分，当乘法操作结束时，输出完成信号。由于阵列结构乘法器和加法器都是组合电路，所以可以在一个时钟周期内实现 4 位乘法和 8 位加法操作。注意，所写的代码中不要包含阵列结构乘法器模块，但是可以使用重载运算符“ * ”。如果 *D* 和 *E* 均为 4 位数，则 D*E 会生成 8 位的积。

(a) 画出此控制器的状态图（10 个状态）。

(b) 写出乘法器的 VHDL 代码，使用两个进程，且所有信号的和数据类型为 unsigned 或 bit。



4.16 (a) 若要实现一个 16 位×16 位的数组乘法器，估计一下所需 AND 门和加法器的数量。

(b) 假设 AND 门的延迟为 t_g，加法器（全加器和半加器）延迟为 t_{ad}，则 16×16 阵列结构乘法器 的最大延迟为多少?

4.17 (a) 一个 2×2 阵列结构乘法器，被乘数和乘数均为 2 位，结果为 4 位。画出此乘法器的结构图。实现此乘法器需要多少个 AND 门、全加器和半加器?

(b) 在(a)的答案中用高亮线标出关键路径（如果存在多条等价路径，则标出其中的一条即可）。

(c) 假设 AND 门延迟 t_g=1 ns，加法器（全加器和半加器）延迟 t_{ad}=2 ns，则 8×8 阵列结构乘法器的最大延迟为多少?

(d) 要想速度与(c)中一样快，一个 8 位×8 位相加–移位乘法器（类似于图 4.25）的时钟频率应为多少?

4.18 图 4.29 的 n×n 阵列结构乘法器有($3n$–4)个加法器延迟和 1 个门延迟，设计一个速度更快的阵列结构乘法器(n > 4)。（提示：不用把进位输出传送到左边的加法器，而是把它送到斜下角的加法器中，以加快关键路径，此种拓扑结构称为“使用进位存储加法器的乘法器”）

4.19 一个有符号（补码）二进制数乘法器的框图如图 4.10 所示。当被乘数为–1/8，乘数为–3/8 时，求每个时钟脉冲过后寄存器 *A* 和 *B* 中的内容。

4.20 在 4.10 节我们扩展了有符号二进制数小数（用补码表示负小数）乘法的计算算法。

(a) 通过计算 1.0111×1.101 说明此算法。

(b) 乘数为 4 位（包括符号），被乘数为 5 位（包括符号）。画出使用此算法实现乘法运算的必要硬件结构框图。

4.21 用 Booth 算法设计有符号二进制数乘法器的程序并加以仿真。所有负数均用补码表示。假设每个数均为 n 位（包含符号位）；用一个 n+1 位寄存器 A 作为累加器，这样符号位在存在溢出的情况下也不会丢失；用一个 n+1 位寄存器 B 存储乘数；用一个 n 位寄存器 C 存储被乘数。Booth 算法的工作流程如下：

1. 把 A 清零。把乘数载入到 B 的高 n 位中，把 B_0 清零，把被乘数载入到 C 中。
2. 测试 B 的低两位 B_1B_0。
   如果 B_1B_0=01，则把 C 加到 A 上（C 扩展为 n+1 位，并用 n+1 位加法器加到 A 上）。
   如果 B_1B_0=10，则把 C 的补码加到 A 上。
   如果 B_1B_0=00 或 $_{11}$，跳过此步。
3. 把 A, B 均向右移位并用符号位填补空位。
4. 重复复步骤 2 和步骤 3 (n−1)次。
5. 积在 A 和 B 中（忽略 B_0）。

例：n=5，计算(−9)×(−13)。

	A	B	B_1B_0	
1. 寄存器置数	000000	100110	10	C = 10111
2. 把 C 的补码加到 A 上	<u>001001</u>			
	001001	100110		
3. A, B 均向右移	000100	110011	11	
3. A, B 均向右移	000010	011001	01	
2. 把 C 的补码加到 A 上	<u>110111</u>			
	111001	011001		
3. A, B 均向右移	111100	101100	00	
3. A, B 均向右移	111110	010110	10	
2. 把 C 的补码加到 A 上	<u>001001</u>			
	000111	010110		
3. A, B 均向右移	000011	101011		

最终结果为 0001110101 = + 117。

(a) 画出 n=8 时此系统的框图。用 9 位寄存器 A 和 B、一个 9 位全加器、一个 8 位补码转换器、一个 3 位计数器和一个控制电路实现此系统。计数器用来记录移位的位数。

(b) 画出控制电路的状态图。当计数器状态为 111 时，在最后移位完毕后返回初始状态（3 个状态即可满足要求）。

(c) 用行为描述方式编写此乘法器的程序。

(d) 用下面的例子对所编写的程序进行仿真并验证结果的正确性（每一对中第二个数为乘数）。

01100110×00110011
10100110×01100110
01101011×10001110
11001100×10011001

4.22 设计一个乘法器：两个16位有符号二进制整数相乘，积为32位。负数用补码表示。使用下面的方法：首先，如果被乘数或乘数为负，则把它们转换为补码，然后进行乘法运算并得到积。如果积为负，则对积进行补码转换。当寄存器载入数据后，乘法运算在16个时钟内完成。

(a) 画出此乘法器的框图。用一个4位计数器记录移位个数（状态15时，寄存器输出信号 K=1）。对所有出现在图中的控制信号和条件信号加以定义。

(b) 使用最少的状态数实现控制电路，并画出控制电路的状态图（3个状态）。当乘法计算完成时，控制电路输出完成信号，并等到 ST=0 时返回到状态0。

(c) 用VHDL行为描述方式，并且不使用控制信号编写一个乘法器程序（如图4.35所示）并检测其正确性。

(d) 使用控制信号，用VHDL行为描述方式编写此乘法器程序（如图4.40所示）并检测其正确性。

4.23 设计一个并行加减器，它可对一个用符号位和数值位构成的8位数进行运算。输入 X, Y 和输出 Z 均由符号位和数值位构成。内部的计算既可以用补码也可以用反码，但是不可以假设 X 和 Y 为补码或反码。如果输入信号 Sub=1，则 $Z=X-Y$，否则 $Z=X+Y$。你所设计的电路必须可以为所有的正负输入组合作加减运算。你可以用以下元件实现此系统：一个8位加法器，一个反码转换器（对输入 Y 取反），第二个反/补码转换器（反码转换器和补码转换器二者选一），一个可以生成控制信号的组合逻辑电路[提示：$-X+Y=-(X-Y)$]，另外如果结果不能用符号位和数值位构成的8位数表示，请给出一个向上溢出信号。

(a) 画出框图。不允许出现寄存器、多路选择器和三态总线。

(b) 写出生成控制信号的逻辑电路的真值表。输入为 Sub, Xs 和 Ys，其中 Xs 为 X 的符号，Ys 为 Y 的符号。

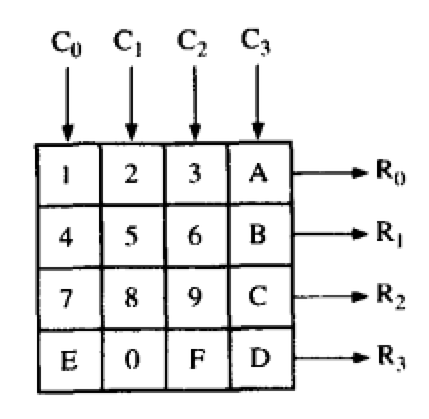
(c) 请解释你是如何定义向上溢出的，并给出正确的表达式。

4.24 一个逻辑电路有4个按键（B_0, B_1, B_2 和 B_3）作为输入。无论何时只要有一个键被按下，防颤抖后电路把此键的二进制键值载入到一个2位寄位器(N)中。如果 B_2 被按下，则寄存器输出为 $N=10_2$。在另一个键被按下前，寄存器将一直保持前一个键值。只使用两个触发器进行去颤抖处理，用一个10位计数器作分频器，并为防颤抖过程提供一个较慢的时钟。当任何键被按下时，信号 Kd 为1。

(a) 画出状态图（2个状态），并生成一个信号，它可以在 Kd=1 时对寄存器载入数据。

(b) 画出模块的逻辑电路图，要求图中显示10位计数器，2位寄存器 N 以及其他所需的门和触发器。

4.25 参照下图设计一个4行4列的键盘扫描器。



(a) 假设一次只能按一个键，输入为 $R_{3\text{-}0}$ 和 $C_{3\text{-}0}$，输出按键对应的二进制数值，例如，如果按下键F，就会返回 $N_{3\text{-}0}$=1111（二进制）或15。求满足条件的数值译码器的逻辑表达式。

(b) 设计一个防颤抖电路，用其检测是否有键被按下。假设在两个时钟周期内开关的抖动脉冲结束，进入正常工作。当按下某一个键时，*K*=1，*Kd* 为防颤抖后的信号。

(c) 为键盘扫描设计一个 SM 图，并且当得到(b)中按键信号后给出一个有效脉冲。

(d) 写出键盘扫描器 VHDL 程序，包含译码器、防颤抖电路和扫描器。

4.26 设计一个无符号二进制数除法器。被除数为 16 位，除数为 8 位，商为 8 位。假设启动信号(*ST*=1)持续一个时钟周期。如果商的位数多于 8 位，则除法器立即停止工作，并输出 *V*=1 以指示出现向上溢出。使用一个 17 位被除数寄存器，并用此寄存器的低 8 位存储商。用一个 4 位计数器和减法移位控制器记录移位的个数。

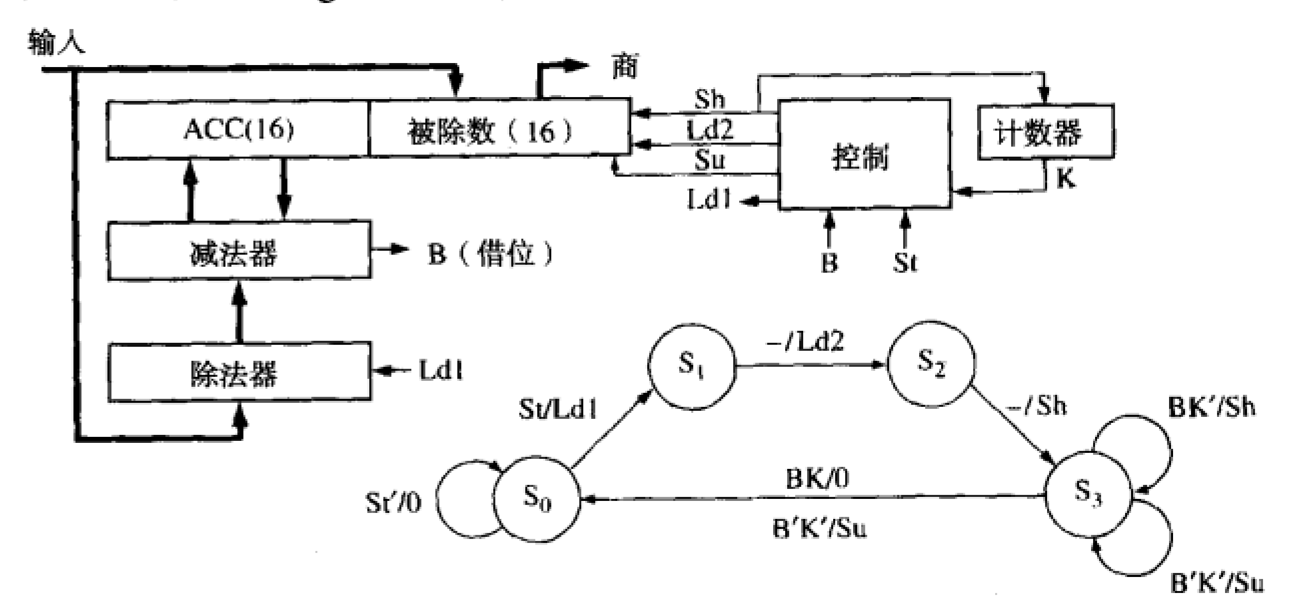
(a) 画出除法器的框图。

(b) 画出减法移位控制器的状态图（3 状态）。

(c) 编写此除法器的 VHDL 程序，使用两个进程，与图 4.40 相似。

(d) 为你设计的除法器编写一个测试平台（与图 4.55 相似）。

4.27 下面是一个无符号二进制数除法器的实现框图和状态图。此除法器可以计算两个 16 位数相除并得到一个 16 位的商。除数可以是 $1\sim(2^{16}-1)$之间的任意数。此除法器只在除数为 0 时产生溢出。控制信号定义如下：*Ld1*：从输入总线上载入除数；*Ld2*：从输入总线上载入被除数，并对 ACC 清零；*Sh*：对 ACC & Divided 左移；*Sn*：把减法器的输出载入到 ACC 中并把较低的商位置 1；当移位 15 次后，*K*=1。写出此除法器的完整的 VHDL 代码，信号的数据类型必须为 unsigned 或 bit，要求程序中要使用两个进程。



4.28 一个除法器的实现框图如下所示。被除数为 8 位，除数为 4 位，商为 4 位。注意 Xi 均左移一位后再输入到减法器。这样移位和减法运算就可以在一个时钟周期内完成（而不是两个时钟周期）。在每个时钟周期内，根据减法器是否借位，决定进行移位操作还是移位相减操作。除法运算必须在寄存器载入数据后 4 个时钟内完成。忽略向上溢出。当开始信号(*St*)为 1 时，寄存器 *X* 和 *Y* 载入数据。假设开始信号(*St*=1)只持续一个时钟。*Sh* 使信号左移并用 0 补空位。*SubSh* 使减法器输出载入到 *X* 的左边部分，同时 *X* 中剩余的位左移。

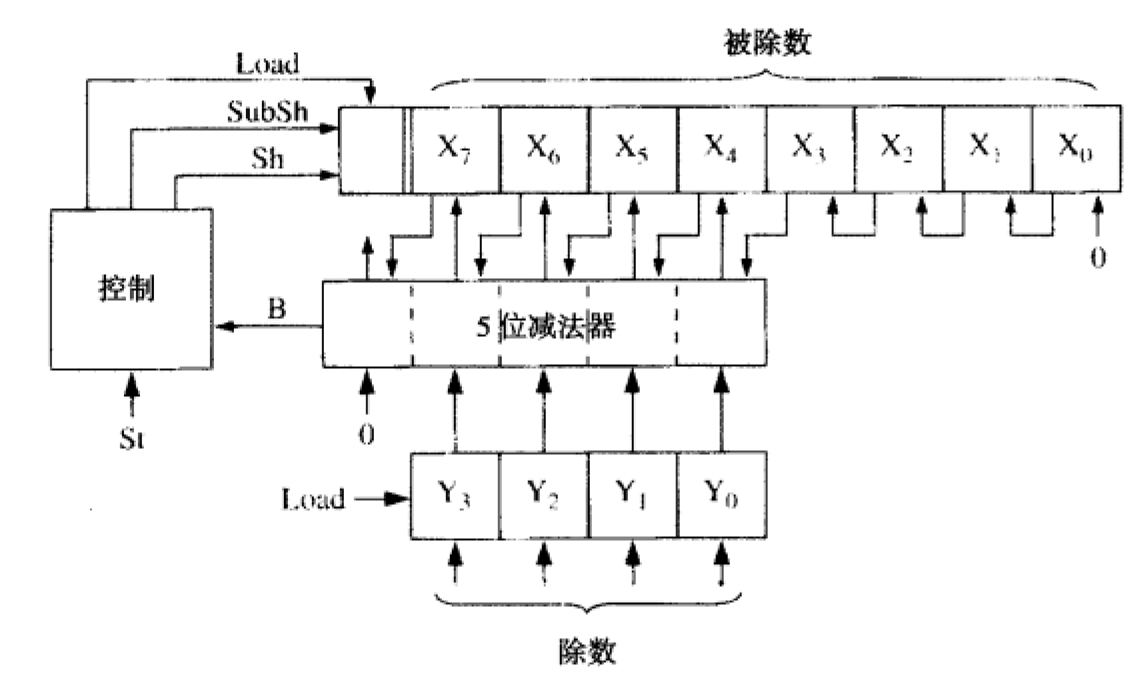
(a) 画出控制器的状态图（5 个状态）。

(b) 完成下面的 VHDL 代码。寄存器和信号的数据类型为无符号类型，这样就可以使用重载运算符了。写出行为描述方式代码，要求只使用一个进程。

```
library IEEE;
use IEEE.numeric_bit.all;
```

```
entity divu is
  port(dividend: in unsigned(7 downto 0);
       divisor: in unsigned(3 downto 0);
       St, clk: in bit;
       quotient: out unsigned(3 downto 0));
end entity divu;

architecture div of divu is
```



4.29 一个老式的雷鸟车在左右两侧各有 3 个尾灯，它们按照特定的方式闪烁来表示左转弯或右转弯。设计一个 Moore 时序电路来控制这些灯。电路有 3 个输入 *LEFT*, *RIGHT* 和 *HAZ*。*LEFT* 和 *RIGHT* 来自司机的转向信号开关且它们不能同时为 1。当 *LEFT*=1 时尾灯按以下顺序点亮：*LA* 亮；*LA* 和 *LB* 亮；*LA*, *LB* 和 *LC* 亮；全灭；然后又开始下一个循环的闪烁序列。当 *RIGHT*=1 时，尾灯 *RA*, *RB* 和 *RC* 的闪烁顺序类似。如果在闪烁过程中，如果司机控制左右转开关使其由左变为右时，电路进入 IDLE（灯全灭）状态，然后再重新亮灯。*HAZ* 由冒险开关给出。当 *HAZ*=1 时，6 个灯同时亮或灭。即使在 *LEFT* 或 *RIGHT* 有效时，*HAZ* 也具有优先权。假设时钟信号频率与闪灯频率相同。

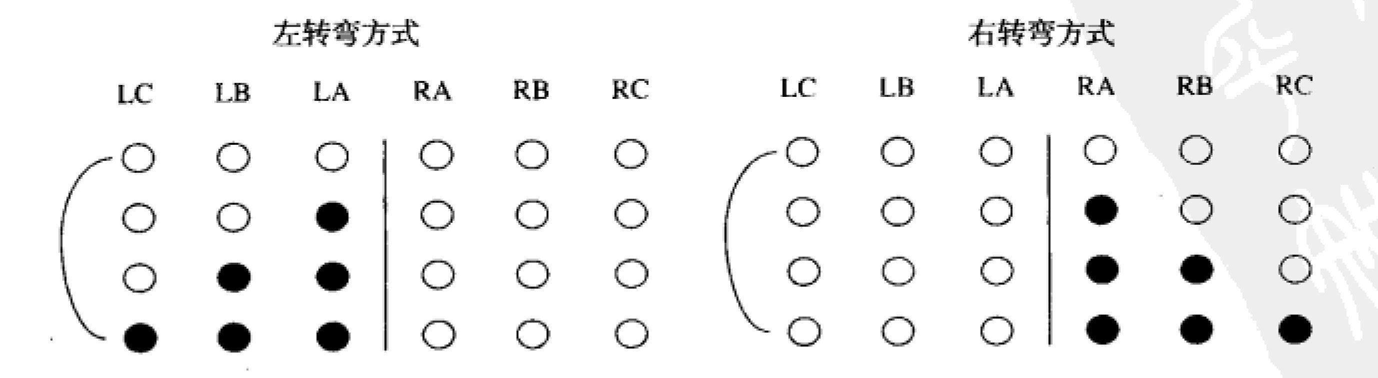
(a) 画出状态图（8 个状态）。

(b) 用 6 个 D 触发器实现此电路，并对状态赋值使每个触发器的输出直接控制一个灯。

(c) 用 3 个 D 触发器实现此电路，并使用导航线选择适当的状态赋值。

(d) 观察(b), (c)中触发器和门电路的替换使用，并对每一个电路设计一个适当的PLA或PLD电路。

(e) 写出(b)电路的 VHDL 代码并仿真。



4.30 设计一个可以控制磁带播放器马达的时序逻辑电路，此逻辑电路具有 5 个输入和 3 个输出，其中 4 个输入为读带机的 4 个控制按键。当播放键被按下时，输入 PL 为 1；当倒带键被按下时，输入 RE 为 1；当快进键被按下时，输入 FF 为 1；当停止键被按下时，输入 ST 为 1。当专用“音乐传感器”在当前磁带位置上检测到音乐时，控制电路的第 5 个输入 M 为 1。当对磁带进行播放、回退、快进操作时，控制电路的三个输出 P, R 和 F 分别为 1，且一次只输出一个信号，当马达停止工作时，所有输出均为 0，如何通过按键控制磁带呢？当播放键被按下时，磁带播放器播放磁带（输出 $P=1$）；如果按下播放键的同时，按下再放开回退键，则回到当前歌曲的起始处播放（当 $M=0$ 时输出 $R=1$）；如果按下播放键，快进键按下再放开，则前进到当前歌曲的末尾处开始播放（当 $M=0$ 时输出 $F=1$）；如果按下回退或快进键的同时没有按下播放键，则对磁带进行前倒带和后倒带；任何时刻只要按下停止键，则播放器停止工作。

(a) 画出此控制器的状态图，你可以假设再任意给定时刻，几个输入键中只有一个被按下。

(b) 写出控制器的 VHDL 代码。



第 5 章　SM 图与微程序

过去对于执行逐步算法或者程序的数字系统，我们经常用状态机来进行控制。在状态图中，我们使用圆圈把各个状态圈起来，并在圆圈上用圆弧表示控制状态机操作的转换条件。除了用状态转换图之外，我们还可以用一种特殊的流程图——状态机图或 SM 图——来描述状态机的行为特性。SM 图也叫做算法状态机图，即 ASM 图。SM 图常用来设计数字系统的控制单元。

本章首先叙述了 SM 图的特性以及它们如何应用于状态机设计，然后举例说明 SM 图在乘法器和掷骰子游戏控制器中的应用。我们根据 SM 图，使用 VHDL 语言对这些系统进行描述，并对 VHDL 代码进行仿真以验证其操作的正确性。然后我们进一步进行设计，并指出如何使用硬件实现 SM 图。最后我们介绍了 SM 图的一种实现技术——微程序。

5.1 状态机流程图

SM 图同软件流程图很相似。在过去的几十年中，流程图在软件设计中非常有用；同样，SM 图在数字系统硬件设计中也非常有用，特别是在行为描述方式的设计中。

相对于状态图来说，SM 图有几个优点。通常来说，通过观察 SM 图会比观察等价的状态图更加容易让人理解数字系统的执行过程。一个适宜的状态图必须遵循一些条件：（1）任何时刻从一个状态出发的转移中，有且只有一条为真；（2）必须为每个输入组合定义唯一的下一状态；而 SM 图可以自动满足这些条件。SM 图还可以直接同硬件实现联系起来。一个给定的 SM 图能转换成几个等价的形式，而且不同的形式自然可以得到不同的实现。因此，设计者可以对 SM 图进行优化和转换，以适应目标技术和实现方式。

SM 图与一般的的流程图不同。建立 SM 图要遵循一定的特定规则。只有遵循了这些规则，SM 图才能与状态图等价，就能直接得出硬件实现方法。

图 5.1 给出了一个 SM 图的三个基本组成部分。系统的状态由一个**状态框**表示，状态框包含一个**状态名**，状态名后面紧跟着一条斜线（/）和一个可选的**输出列表**。状态赋值后，在状态框外部的上方会放一个状态码。**判决框**由一个菱形的符号表示。此符号由两个分支组成：条件成立分支和条件不成立分支。放在框内的条件是一个布尔表达式，此表达式用于计算并决定执行哪一个分支。条件输出框（框的边缘为曲线）含有一个条件输出列表。条件的输出取决于系统的状态和输入。

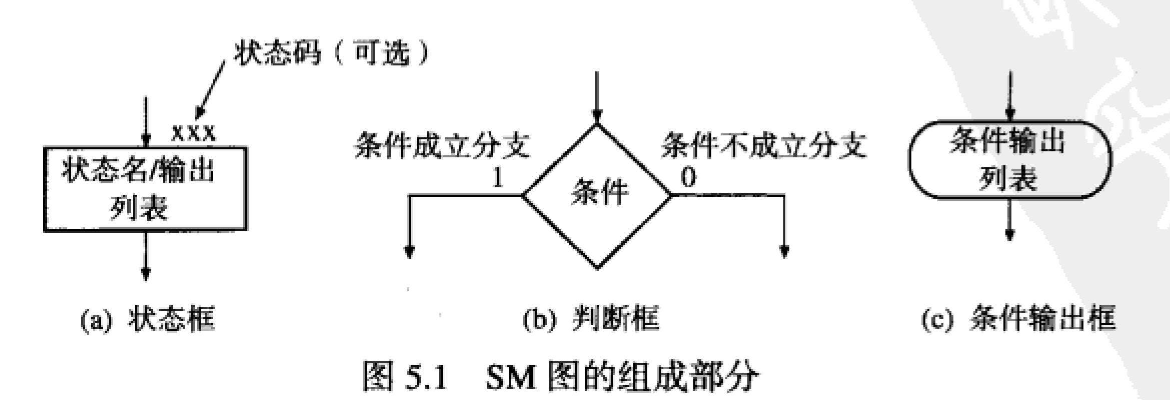


图 5.1　SM 图的组成部分

一个 SM 图由 SM 模块构成。每个 SM 模块（图 5.2）仅含有一个状态框和多个条件判断框以及与状态相关的条件输出框。一个 SM 模块仅含有一个输入通路以及一个或多个输出通路。每个 SM 模块描述状态机在一个状态时序里的运行。当数字系统进入到给定的 SM 模块相关的状态时，状态框里输出列表上的输出为真。根据判决框里的条件，判决该执行 SM 模块里的哪一条路径（或多条路径）。当沿着这样的路径到一个条件输出框时，相应的条件输出为真；其他没有沿此路径到达的输出为假。贯穿一个 SM 模块，从入口到出口的路径称为一条链路。

以图 5.2 为例，当进入状态 S_1 时，输出 Z_1 和 Z_2 变为 1。如果输入 $X_1 = 0$，则 Z_3 和 Z_4 也变为 1。如果 $X_1 = X_2 = 0$，则状态机会在当前状态结束时通过输出通路 1 进入下一个状态；另一种情况是，如果 $X_1 = 0, X_3 = 0$，则输出 Z_5 为 1，状态机就会通过输出通路 3 由当前状态进入下一个状态，由于接路没有通过 Z_3 和 Z_4，所以默认 $Z_3 = Z_4 = 0$。

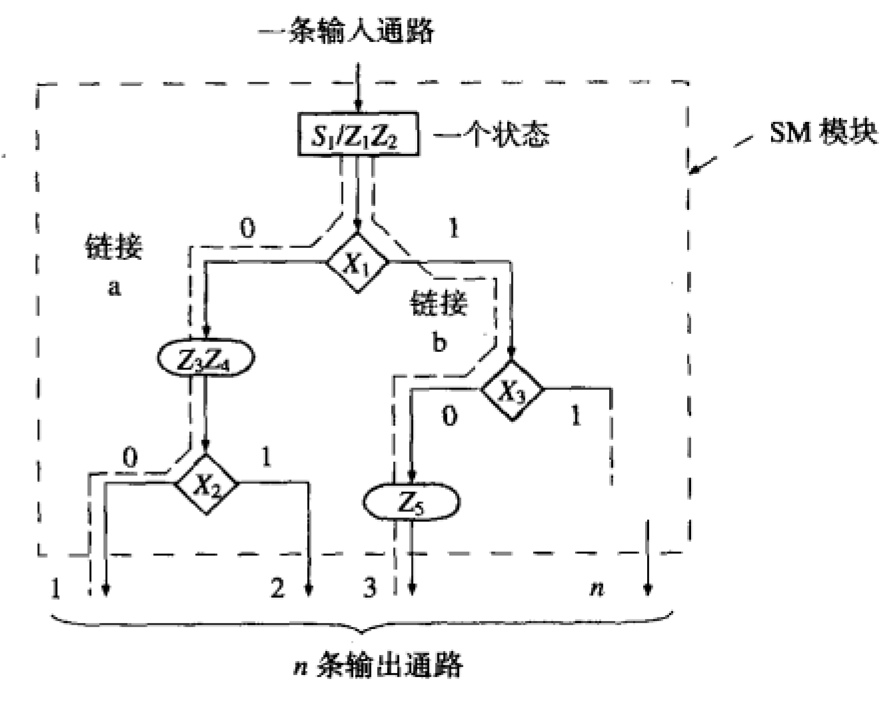


图 5.2　SM 模块实例

一个给定的 SM 模块通常可以用几种不同的形式描述。图 5.3 给出了两个等价的 SM 模块。在(a)和(b)中，如果 X_1=0，则输出 $Z_2 = 1$；如果 $X_2 = 0$，则下一个状态为 S_2；如果 $X_2 = 1$，则下一个状态为 S_3。正如此例所示，输入的检测顺序能够影响 SM 图的复杂度。

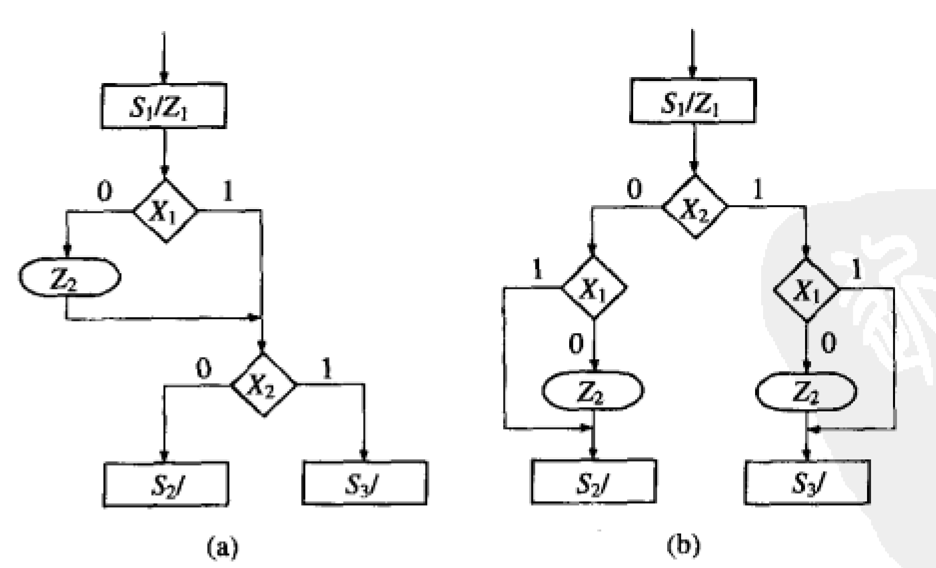


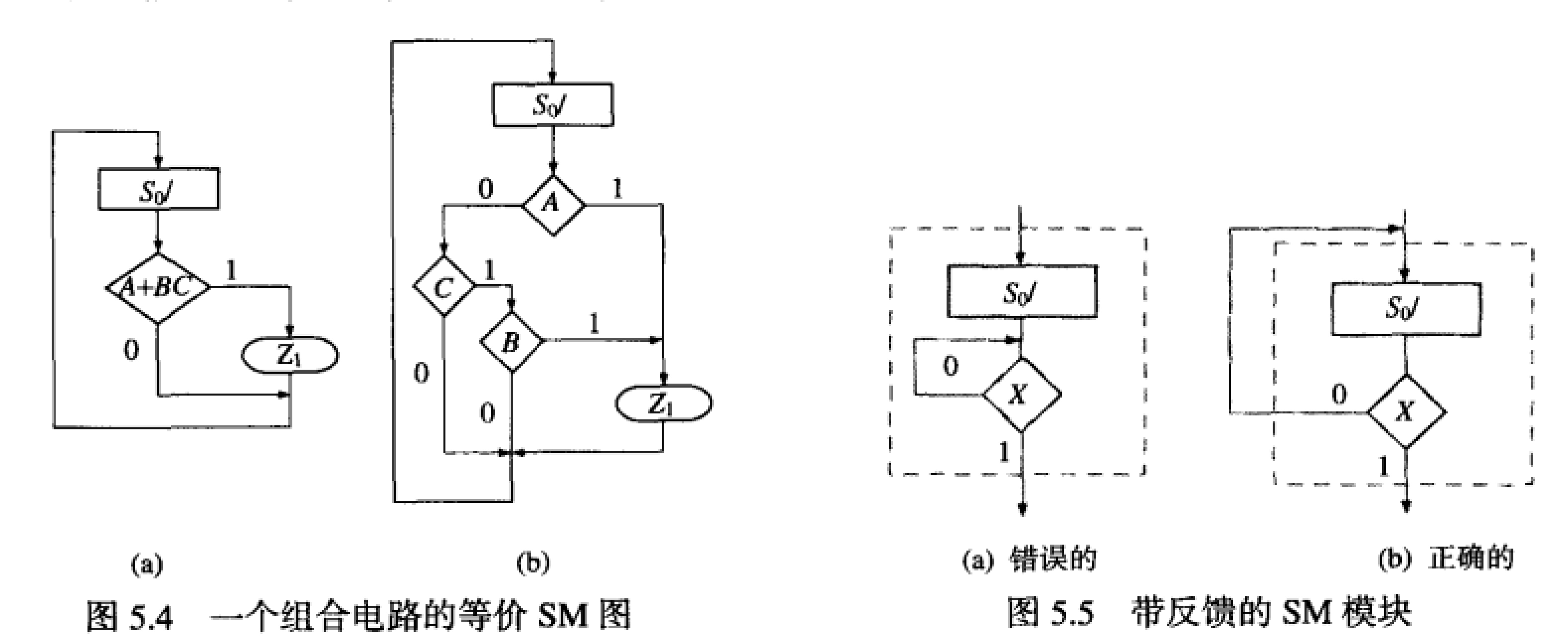
图 5.3　等价 SM 模块

图 5.4(a)和(b)的 SM 图各自代表一个组合电路——因为只有一个状态，没有状态转换。如果 $A + BC = 1$，则输出 $Z_1 = 1$；否则 $Z_1 = 0$。图 5.4(b)给出了一个等价的 SM 图，在这个图里对输入变量进行单独的检测。如果 $A = 1$ 或者 $A = 0, B = C = 1$，则输出 $Z_1 = 1$。因此有

$$Z_1 = A + A'BC = A + BC$$

这与图 5.4(a)中的 SM 图所实现的输出函数是一致的。

建立一个 SM 模块要遵循一定的规则。首先，对于输入变量的每一个有效组合，只能有一个输出通路。这是必须的，因为每个允许的输入组合只能产生一个下一个状态。其次，在一个 SM 模块里，不能有“内部反馈”。图 5.5 给出了描述带反馈的 SM 模块的正确方式和错误方式。



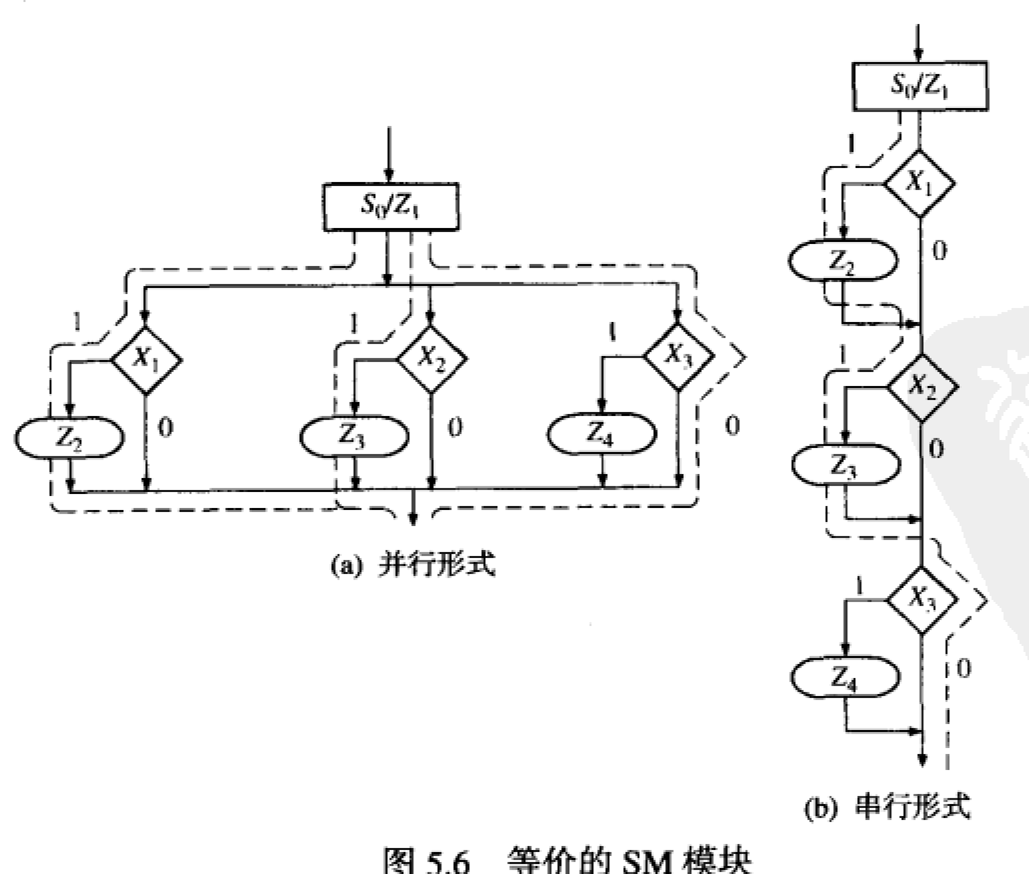
(a)　(b)

图 5.4　一个组合电路的等价 SM 图

(a) 错误的　(b) 正确的

图 5.5　带反馈的 SM 模块

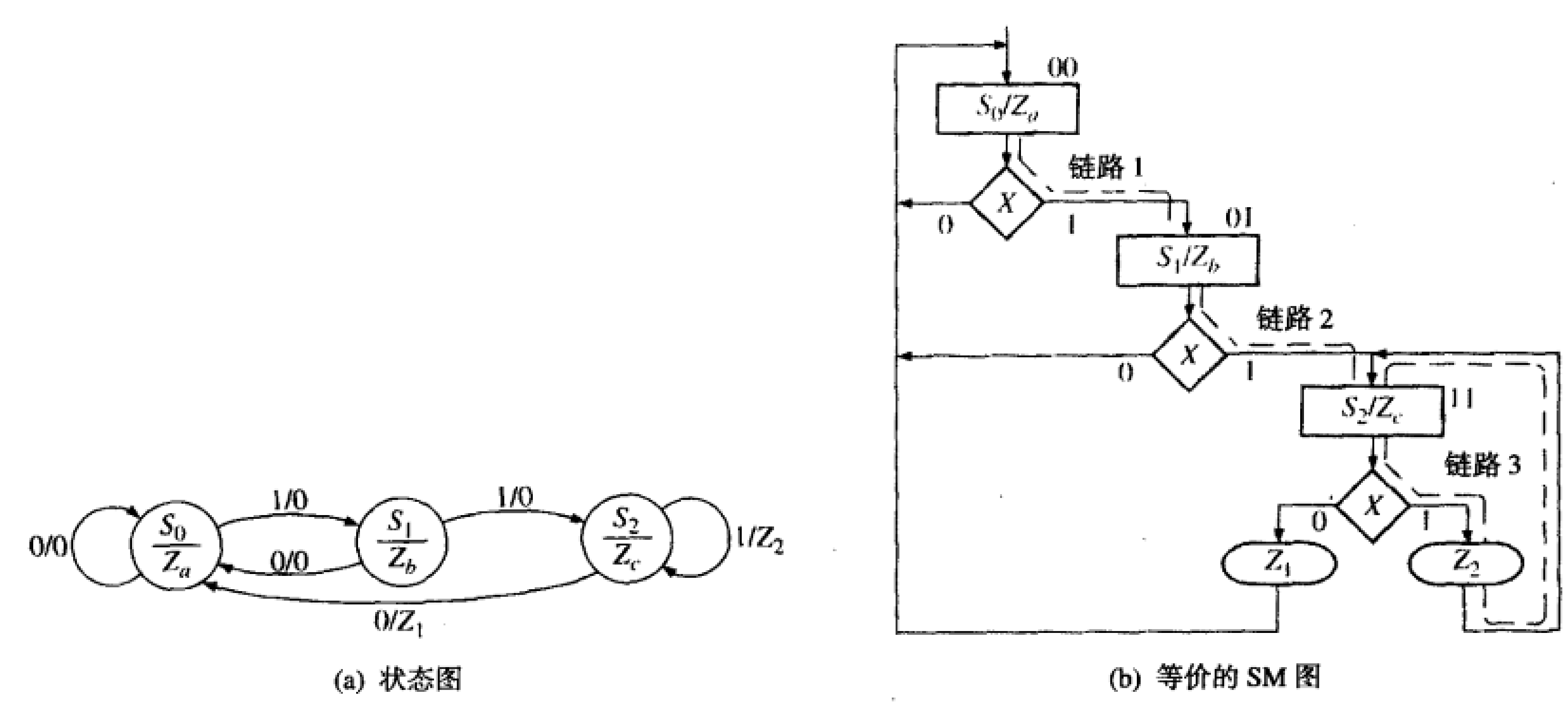
正如图 5.6(a)中所显示的，一个 SM 模块可以有多个并行的路径通往同一个输出通路，并且不止一个路径可以在同一时间被选通。例如，如果 $X_1 = X_2 = 1$ 且 $X_3 = 0$，用虚线标出的链路都是被选通的，输出 Z_1, Z_2 和 Z_3 都将是 1。虽然图 5.6(a)在串行计算机里是一个不合法的流程图，但是它在状态机上实现是没有问题的。状态机可以有一个多输出的电路，可以在同一时间产生 Z_1, Z_2 和 Z_3。图 5.6(b)给出了一个与图 5.6(a)等价的串行 SM 模块。在这个串行模块的入口与出口之间只可能存在一条链路。对于任何输入组合，其输出与等价的并行结构的输出相同。$X_1 = X_2 = 1$ 且 $X_3 = 0$ 的链路用虚线表示，此链路的输出为 Z_1, Z_2 和 Z_3。不管 SM 模块是串行方式的还是并行方式的，所有的检测都是在一个时钟周期内发生的。在下文中，我们只使用串行的 SM 图。



(a) 并行形式　(b) 串行形式

图 5.6　等价的 SM 模块

将时序机的状态图转化为与之等价的 SM 图是很容易的。图 5.7(a)中的状态图中 Moore 输出和 Mealy 输出都有。等价 SM 图有三个模块，每个模块对应一个状态。Moore 输出（Z_a, Z_b, Z_c）放置在状态框中——因为它们不取决于输入。Mealy 输出（Z_1, Z_2）则放置在条件输出框中——因为它们由状态和输入决定。在这个例子中，因为只需检测一个输入变量，所以每个 SM 模块只有一个判决框。对于状态图和 SM 图，在状态 S_2 时 Z_c 都是 1。在状态 S_2 时，如果 $X = 0, Z_1 = 1$，则下一个状态为 S_0。如果 $X = 1, Z_2 = 1$，则下一个状态为 S_2。我们在状态框边上加上了状态赋值（$S_0 = 00, S_1 = 01, S_2 = 10$）。



(a) 状态图 (b) 等价的 SM 图

图 5.7 状态图向 SM 图的转换

图 5.8 给出了当输入序列为 $X = 1, 1, 1, 0, 0, 0$ 时，图 5.7 中 SM 图的时序图。在这个例子中，所有状态均在时钟的上升沿之后立即改变。因为 Moore 输出（Z_a, Z_b, Z_c）取决于状态，所以它们只能在某个状态变化后立刻改变。而 Mealy 输出（Z_1, Z_2）可以在某一状态变化之后或某一输入变化之后立刻改变。不论如何，所有的输出都可以在时钟有效沿都应具有正确的值。

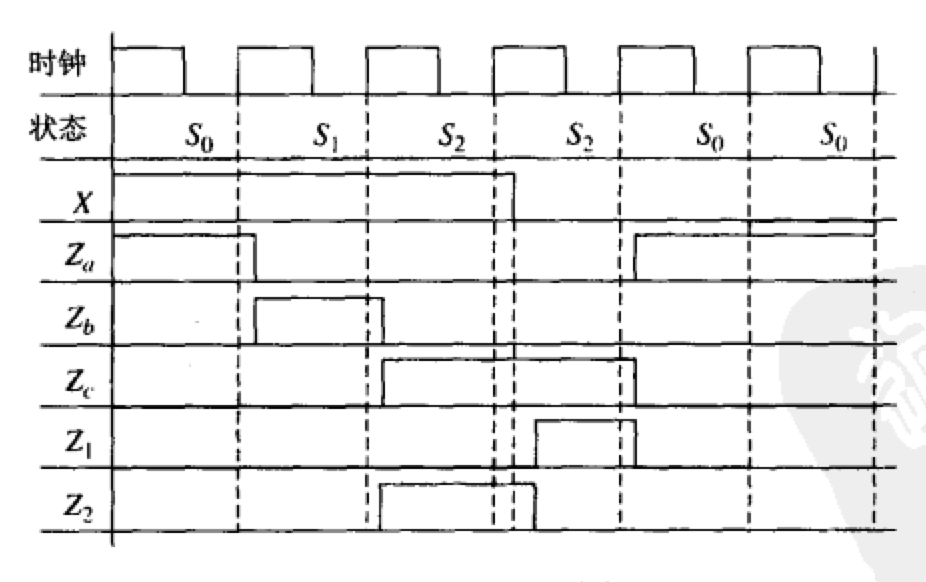


图 5.8 图 5.7 的时序

5.2 SM 图的推导

一个时序控制电路的 SM 图的推导方法与状态图的推导方法相似。我们先画出要控制的系统的方框图，其次定义控制电路的输入和输出信号。这样我们就可以建立一个 SM 图，用于检测输

入信号，并且产生正确的输出信号序列。在本节中，我们给出两个例子说明如何推导得到 SM 图。

5.2.1　二进制乘法器

第一个例子是图 4.25 和图 4.28(a)所示二进制乘法器的控制器的 SM 图。加法移位控制器产生所需要的加法和移位信号序列。计数器计算移位次数，并在最后一次移位发生前输出 K=1。乘法器的控制器（图 5.9）的 SM 图与图 4.28(c)的状态图基本上是一致的。在状态 S_0，当开始信号 S_t 为 1 时，寄存器置数。在 S_1 对乘数位 M 进行检测。如果 $M = 1$，则生成一个"相加"信号，下一状态为 S_2。如果 $M = 0$，则生成"移位"信号，并对 K 进行检测。如果 $K = 1$，则此次移位为最后一次移位，并且下一个状态为 S_3。因为在加操作后必须接着一个移位操作，所以在 S_2 产生一个移位信号。如果 $K = 1$，则电路在最后一次移位时转到 S_3，否则下一状态为 S_1。在 S_3 生成完成信号（Done）。

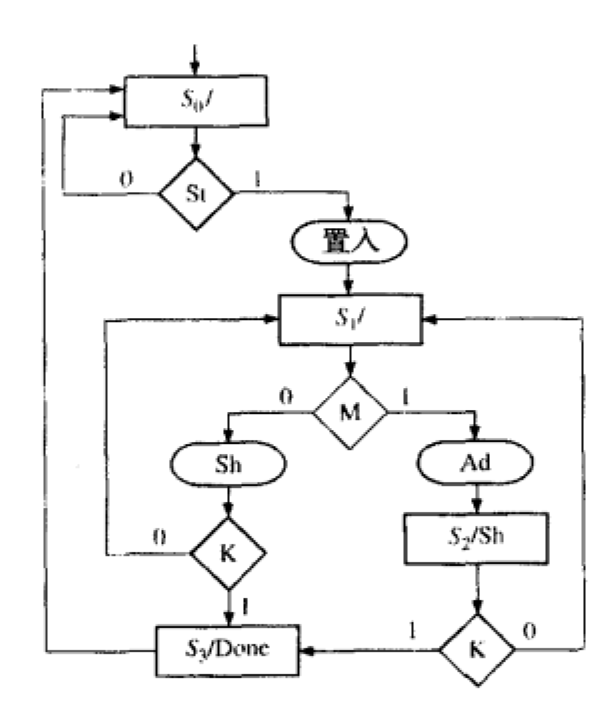


图 5.9　二进制乘法器的 SM 图

我们可以直接将 SM 图转化成 VHDL 程序。使用 case 语句设定在每个状态中发生的事项，每个 if（或者 else if）语句直接对应一个条件框。图 5.10 给出了图 5.9 中 SM 图的 VHDL 代码，此代码中使用了两个进程。第一个进程描述了电路的组合逻辑部分，第二个进程在时钟上升沿到来时对状态寄存器进行更新。信号 *Load*, *Sh* 和 *Ad* 必须在合适的状态中生成，并且在状态发生改变时关闭。为了方便地实现次操作，我们可以在进程初始化时就把这些信号都设置为零。此 VHDL 代码只对控制器进行了模拟。在代码的结构体中，假设已存在加法器和移位器（移位寄存器），并生成恰当的信号实现寄存器置数、加法操作和移位操作。

```
entity Mult is
  port(CLK, St, K, M: in bit;
       Load, Sh, Ad, Done: out bit);
end Mult;

architecture SMbehave of Mult is
signal State, Nextstate: integer range 0 to 3;
begin
  process(St, K, M, State)                 -- start if state or inputs change
  begin
    Load <= '0'; Sh <= '0'; Ad <= '0'; Done <= '0';
    case State is
      when 0 =>
        if St = '1' then                   -- St (state 0)
          Load <= '1';
          Nextstate <= 1;
        else Nextstate <= 0;               -- St'
        end if;
      when 1 =>
        if M = '1' then                    -- M (state 1)
          Ad <= '1';
          Nextstate <= 2;
        else                               -- M'
          Sh <= '1';
          if K = '1' then Nextstate <= 3;  -- K
          else Nextstate <= 1;             -- K'
          end if;
        end if;
```

图 5.10　乘法器控制器（图 5.9 的 SM 图）的行为描述方式 VHDL 代码

```
      when 2 =>
        Sh <= '1';                          -- (state 2)
        if K = '1' then Nextstate <= 3;     -- K
        else Nextstate <= 1;                -- K'
        end if;
      when 3 =>
        Done <= '1';                        -- (state 3)
        Nextstate <= 0;
    end case;
  end process;
  process(CLK)
  begin
    if CLK = '1' and CLK'event then
      State <= Nextstate;                   -- update state on rising edge
    end if;
  end process;
end SMbehave;
```

图 5.10（续） 乘法器控制器（图 5.9 的 SM 图）的行为描述方式 VHDL 代码

5.2.2 掷骰子游戏

作为建立 SM 图的第二个例子，我们来设计一个电子掷骰子游戏。在美国，这个游戏被称为双骰子赌博游戏。此游戏使用两个骰子，每次每个骰子均为 1～6 中的一个数字。我们用两个计数器来模拟骰子的滚动，每个计数器的计数序列为 1, 2, 3, 4, 5, 6, 1, 2, …。因此，当“掷”骰子后，两个计数器所得值之和将在 2～12 之间。这个游戏的规则如下：

1. 第一次掷“骰子”后，如果和数为 7 或者 11，则玩家赢。如果和数为 2, 3 或者 12，则玩家输。否则，玩家第一次“掷”出来的和数作为“分数”，而且玩家必须再掷一次骰子。
2. 在第二次或者后续掷骰子时，如果得到的和数与他得到的“分数”相等，则玩家赢。如果和数为 7，则玩家输。否则，玩家必须再次掷骰子直到赢或者输。

图 5.11 给出了掷骰子游戏的实现方框图如图 5.11 所示。掷骰子游戏的输入来自于两个按钮：*Rb*（投掷按钮）和 *Reset*（复位按钮）。*Reset* 复位按钮用来对新游戏初始化开始一个新游戏。当按下投掷按钮时，骰子计数器会高速计算迅速滚动，所以无法在显示器上读出数值。当松开按钮时，两个计数器的数值就会显示出来。

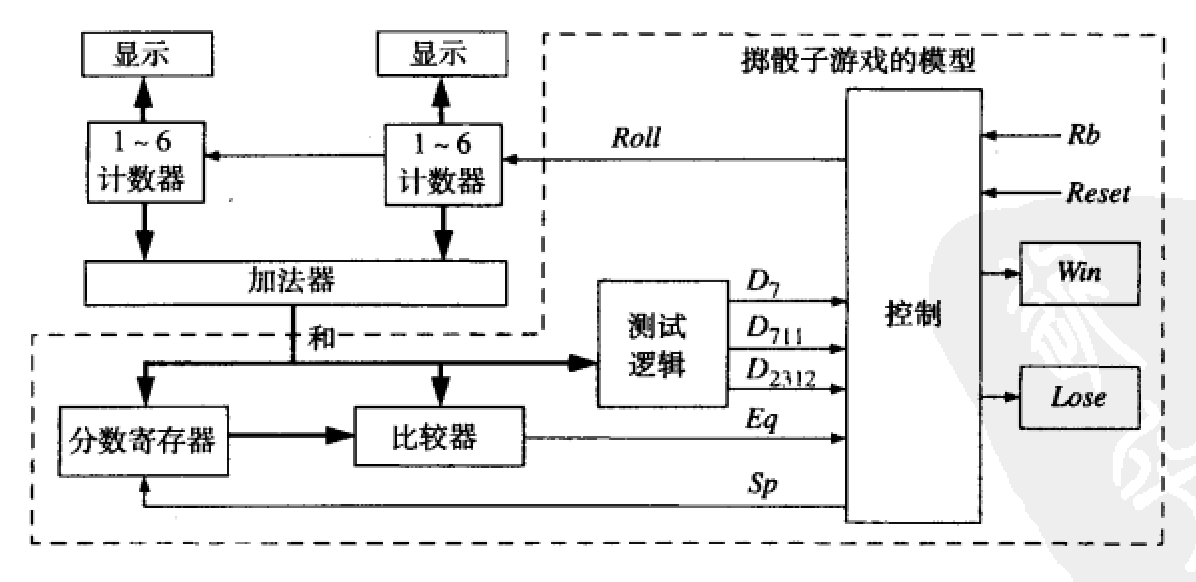


图 5.11 掷骰子游戏的方框图

图 5.12 给出了掷骰子游戏的流程图。当“投掷”了骰子结束后，对所得的和数进行测试检测。如果和数为 7 或者 11，则玩家赢胜出。如果和数为 2, 3 或 12，则玩家失败输。否则，和数被存储在分数指针寄存器中，然后玩家需要再“掷”一次。如果新得出的的和数等于“分数”，则玩家赢。

与指针寄存器存储的数值相等，则玩家胜出。如果和数为 7，则玩家输，否则玩家需要再“重新投掷”一次。当玩家输了或者赢了之后，他必须按下 *Reset* 按钮才能开始新一轮的游戏。我们假设按钮的去抖动和 *Rb* 的变化与时钟能完全同步。去抖动和同步方法在第 4 章中已经讨论过了。

从图 5.11 给出的方框图中可以看出，掷骰子游戏的组成部分包括一个加法器用于将两个计数器的输出相加、一个用于记录分数的寄存器、用于判断输赢的测试逻辑和一个控制电路。控制电路的输入信号的定义如下：

如果骰子的和数为 7，则 $D_7=1$。
如果骰子的和数为 7 或者 11，则 $D_{711}=1$。
如果骰子的和数为 2, 3 或者 12，则 $D_{2312}=1$。
如果骰子的和数与分数寄存器中的分数相等，则 $Eq=1$。
当投掷按钮被按下时，$Rb=1$。
当 *Reset* 按钮被按下时，$Reset=1$。
控制电路的输出定义如下：
$Roll=1$：开启计数器。
$Sp=1$：将和数存储在分数寄存器中。
Win（赢）= 1：点亮 Win 灯。
Lose（输）= 1 点亮 Lose 灯。

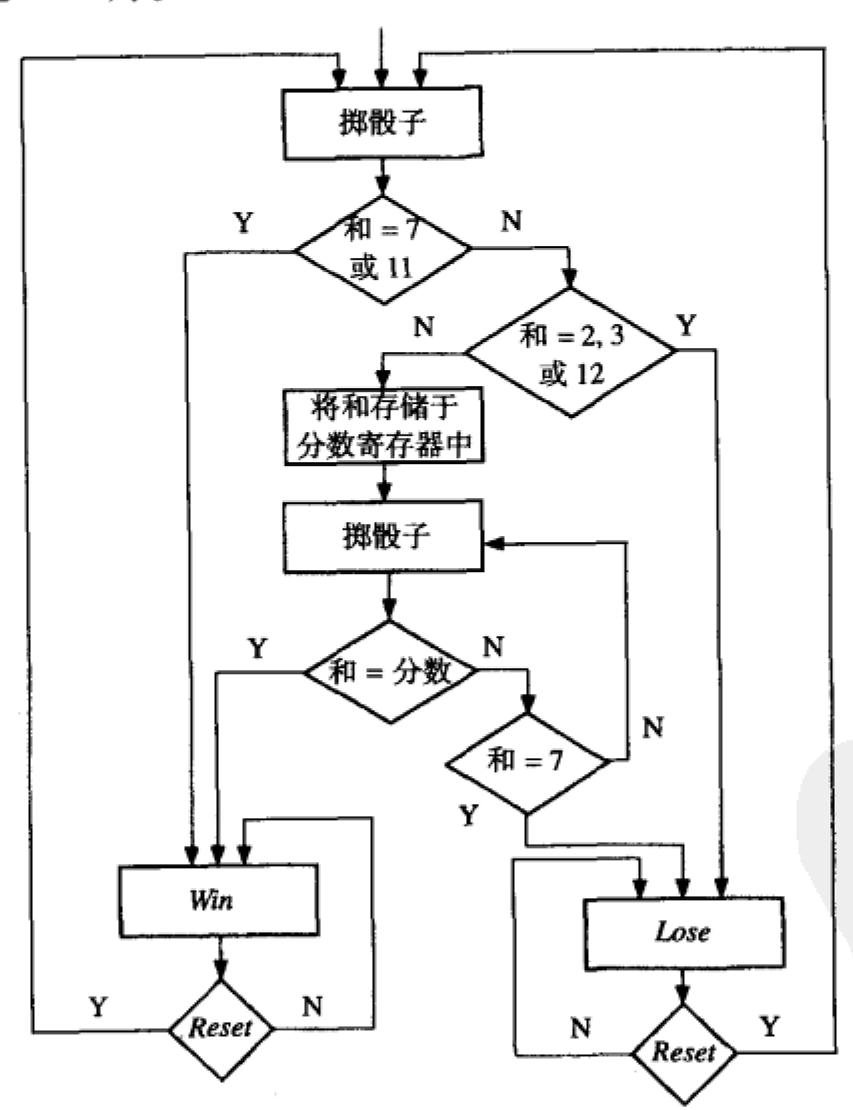


图 5.12 骰子游戏流程图

Rb 和 *Roll* 两个信号看似相同，但它们是不同的。当我们使用电子骰子计数器时，*Roll* 信号使计数器继续计数；而 *Rb* 是按键信号，表示可以掷骰子了。因此，*Rb* 是控制电路的输入信号，而 *Roll* 则为控制电路的输出信号。当控制电路处于等待新一轮“掷”骰子的状态时，无论何时，只

要按下投掷按钮（即 *Rb* 被激活），控制电路就生成 *Roll* 信号送给电子骰子。

现在我们可以用定义的控制信号将掷骰子游戏的流程图转换成控制电路的 SM 图了。图 5.13 给出了所得的 SM 图。

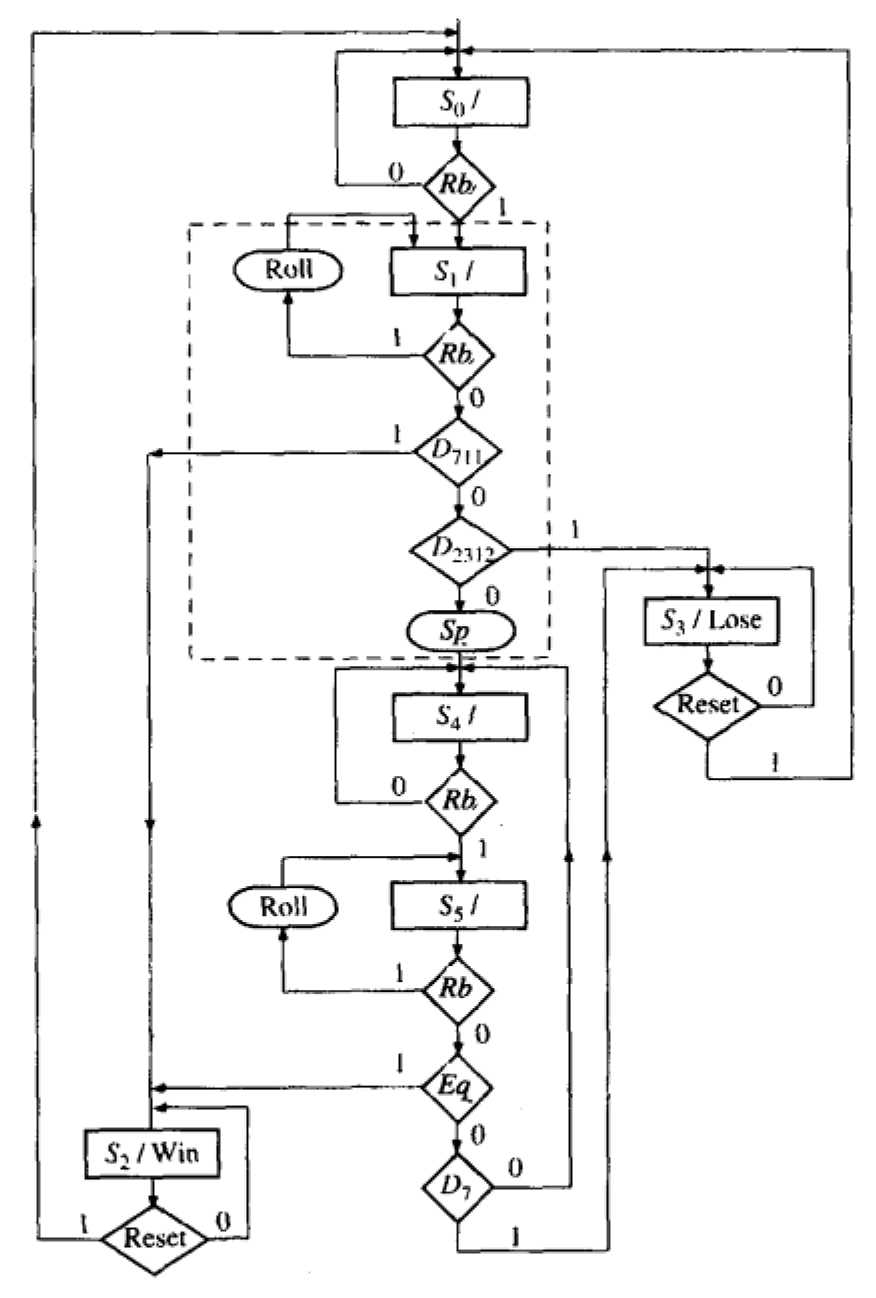


图 5.13 掷骰子游戏的 SM 图

在 *Rb* 按钮被按下（*Rb*=1）前，控制电路将会在状态 S_0 等待。只要 *Rb* = 1，骰子计数器不停地计数，并进入状态 S_1。只要投掷按钮被释放（*Rb*=0），则对 D_{711} 进行检测。如果和数为 7 或 11，则电路进入状态 S_2，并点亮 Win 灯，否则对 D_{2312} 进行检测；如果和数为 2, 3 或 12，则电路进入状态 S_3，并点亮 *Lose* 灯。否则，信号 *Sp* 变为 1，和数被存储在分数寄存器中，然后进入状态 S_4，等待玩家再次“掷骰子”。在状态 S_5 中，当松开投掷按钮 *Rb* 后，如果 *Eq* = 1，则和数等于分数并且进入状态 S_2 表示玩家赢。如果 D_7 = 1，则表示和数为 7，并且电路进入到状态 S_3 表示玩家输。否则，控制电路返回到状态 S_4，让玩家再次掷骰子。当在状态 S_2 和状态 S_3 时，按下 *Reset* 按钮游戏将被重新设置为状态 S_0。

除了使用 SM 图外，我们还可以由流程图建立一个等价的状态图。图 5.14 给出了一个掷骰子游戏控制器的状态图。状态图与 SM 图有一样的状态、输入和输出。弧线上的标注按照 4.5 节中状态图的规则来进行标注。这样，从状态 S_1 出去的弧线上标识了 Rb, $Rb'D'_{711}$, $Rb'D'_{711}D_{2312}$ 和 $Rb'D'_{711}D'_{2312}$。

在进行下一步设计之前，我们首先要验证 SM 图（或状态图）的正确性。我们先给出 SM 图的行为描述 VHDL 代码，然后再编写一个测试平台对投掷骰子的过程进行模拟。首先我们编写一

个掷骰子游戏模块，其包含控制电路、分数寄存器和比较器（见图 5.11）。然后，我们再加入计数器和加法器，这样我们就能对整个掷骰子游戏进行模拟了。

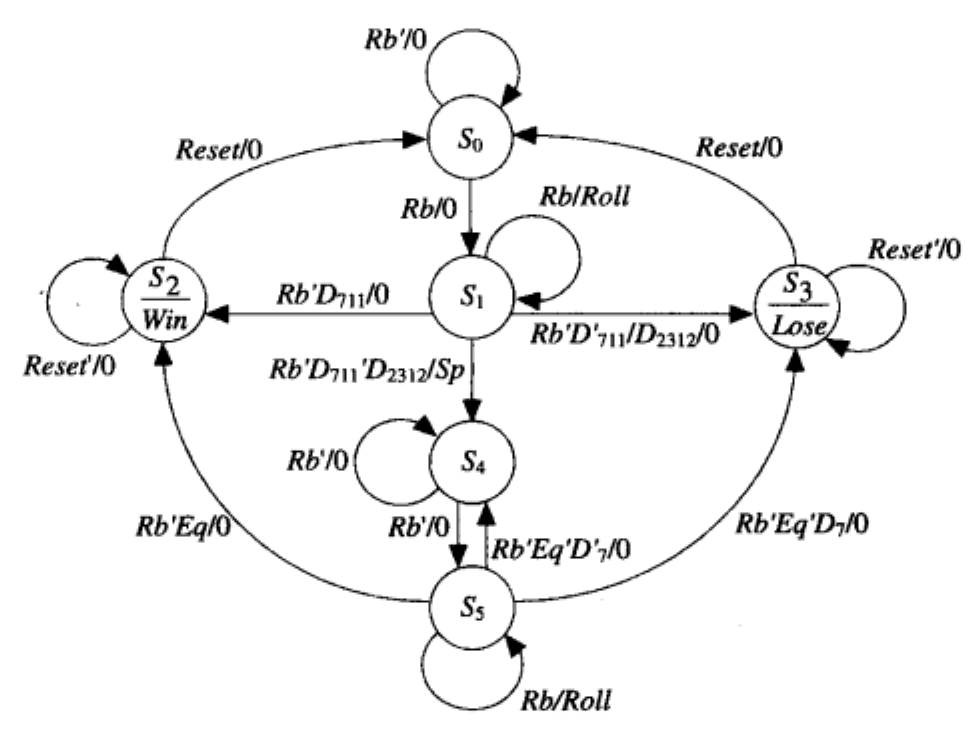


图 5.14　掷骰子游戏控制器的状态图

图 5.15 给出的掷骰子游戏的 VHDL 代码与图 5.13 的 SM 图直接对应。第一个进程（process）中的 case 语句用来进行状态检测，在每个状态中用 if-then-else（或 elsif）嵌套语句实现此测试。在状态 1 中，当 *Rb* 为 1 时，滚动信号开启。如果所有的条件均不满足要求，则 *Sp* 置 1，并且下一状态为 4。在第二个进程（process）中，当时钟上升沿到来后进行状态更新，并且如果 *Sp* 为 1，则和数被存储在分数寄存器中。

下面我们准备测试掷骰子游戏的行为模块。由于在初始测试中很难把生成任意数列的计数器加入到测试平台中，所以我们要设定一个骰子滚动序列，要求通过此序列，我们能够对 SM 图中所有的路径进行测试。我们可以准备一个能够为 *Rb*, *Sum* 和 *Reset* 生成数据序列的仿真命令文件。这需要对时序进行仔细分析，并确保输入信号在恰当的时间发生改变。另外还有一种测试掷骰子游戏的更好的方法，即设计一个 VHDL 测试平台用于监视骰子游戏模块的输出，从而给出响应的输入序列。

```
entity DiceGame is
  port(Rb, Reset, CLK: in bit;
       Sum: in integer range 2 to 12;
       Roll, Win, Lose: out bit);
end DiceGame;

architecture DiceBehave of DiceGame is
signal State, Nextstate: integer range 0 to 5;
signal Point: integer range 2 to 12;
signal Sp: bit;
begin
  process(Rb, Reset, Sum, State)
  begin
    Sp <= '0'; Roll <= '0'; Win <= '0'; Lose <= '0';
    case State is
      when 0 => if Rb = '1' then Nextstate <= 1; end if;
      when 1 =>
        if Rb = '1' then Roll <= '1';
        elsif Sum = 7 or Sum = 11 then Nextstate <= 2;
        elsif Sum = 2 or Sum = 3 or Sum = 12 then Nextstate <= 3;
        else Sp <= '1'; Nextstate <= 4;
        end if;
      when 2 => Win <= '1';
        if Reset = '1' then Nextstate <= 0; end if;
      when 3 => Lose <= '1';
        if Reset = '1' then Nextstate <= 0; end if;
      when 4 => if Rb = '1' then Nextstate <= 5; end if;
      when 5 =>
```

图 5.15　掷骰子游戏控制器行为描述 VHDL 程序

```
        when 4 => if Rb = '1' then Nextstate <= 5; end if;
        when 5 =>
          if Rb = '1' then Roll <= '1';
          elsif Sum = Point then Nextstate <= 2;
          elsif Sum = 7 then Nextstate <= 3;
          else Nextstate <= 4;
          end if;
      end case;
    end process;

    process(CLK)
    begin
      if CLK'event and CLK = '1' then
        State <= Nextstate;
        if Sp = '1' then Point <= Sum; end if;
      end if;
    end process;
end DiceBehave;
```

图 5.15（续） 掷骰子游戏控制器行为描述 VHDL 程序

图 5.16 中骰子游戏模块（DiceGame）被连接到 GameTest 测试模块上，GameTest 测试模块应具有以下功能：

1. 为 *Rb* 信号提供初值。
2. 当 DiceGame 给出响应信号 *Roll* 时，给出一个表示两个骰子和数的 *Sum* 信号。
3. 如果 DiceGame 没有产生 *Win*（赢）信号或 *Lose*（输）信号，则重复步骤 1 和步骤 2，并再次滚动骰子。
4. 当检测到 *Win*（赢）信号或 *Lose*（输）信号时，生成一个 *Reset*（复位）信号，并且重新开始游戏。

图 5.17 给出了 GameTest 测试模块的 SM 图。*Rb* 在状态 T_0 中生成。当 DiceGame 检测到 *Rb* 时，测试模块进入到状态 S_1，并生成 *Roll* 信号。当 GameTest 检测到 *Roll* 时，就从 *Sumarray(i)*中读出表示下次骰子滚动的和数 *Sum*，并且对 *i* 加 1。当到达状态 T_1 时，*Rb* 变为 0，DiceGame 到达状态 S_2, S_3 或 S_4，并且 GameTest 到达 T_2。在状态 T_2 中，检测输出是否为 *Win*（赢）或 *Lose*（输）。如果检测到 *Win* 信号或 *Lose* 信号，则在下次骰子滚动前生成 *Reset* 信号。骰子滚动 *N* 次后，GameTest 到达状态 T_3，并且不再做任何操作。

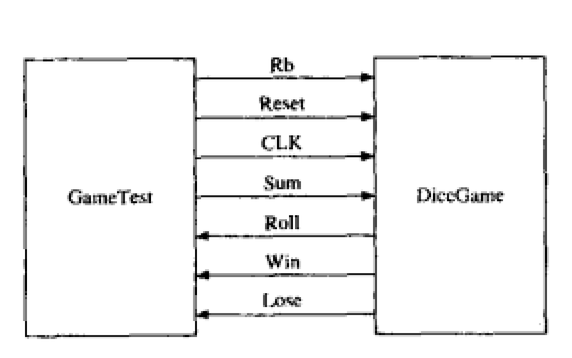


图 5.16 掷骰子游戏的测试平台

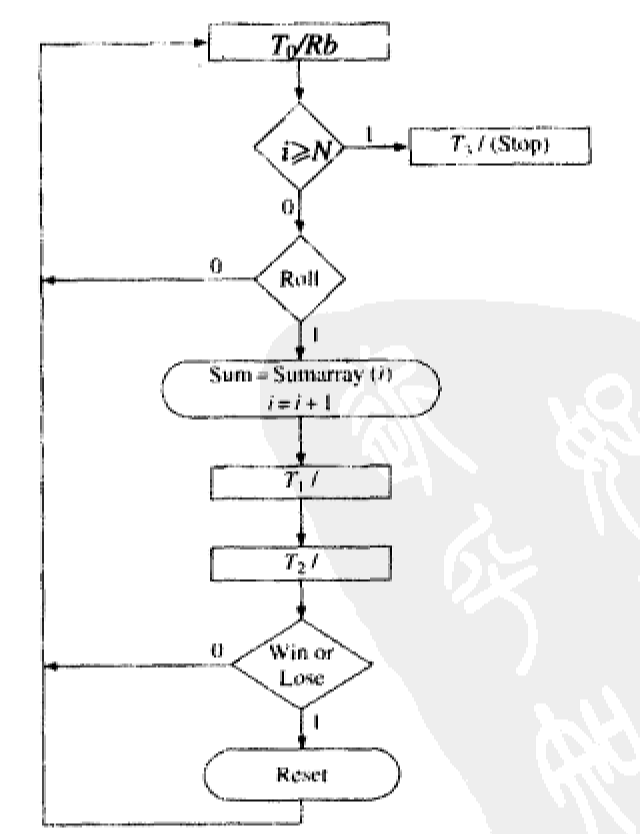


图 5.17 掷骰子游戏测试平台的 SM 图

GameTest（图 5.18）实现了 GameTest 模块的 SM 图。它包含一个测试数据数组、一个生成时钟的并发语句和两个进程。无论何时，只要 *Rol*, *Win*, *Lose* 或 *Tstate*（下一状态）信号发生改变，则第一个进程就生成 *Rb*, *Reset* 和 *Tnext* 信号。第二个进程对 *Ttaste*（GameTest 的状态）进行更新。当仿真器启动时，对于骰子的每次滚动，我们只想显示一条输出通路。因此，我们又添加了一个信号 *Trig1*。每次只要一进入状态 T_2，*Trigl* 就发生了改变。

```
entity GameTest is
  port(Rb, Reset: out bit;
       Sum: out integer range 2 to 12;
       CLK: inout bit;
       Roll, Win, Lose: in bit);
end GameTest;
architecture dicetest of GameTest is
signal Tstate, Tnext: integer range 0 to 3;
signal Trig1: bit;
type arr is array(0 to 11) of integer;
constant Sumarray:arr := (7, 11, 2, 4, 7, 5, 6, 7, 6, 8, 9, 6);
begin
  CLK <= not CLK after 20 ns;
  process(Roll, Win, Lose, Tstate)
  variable i: natural;                     -- i is initialized to 0
  begin
    case Tstate is
      when 0 => Rb <= '1';                 -- wait for Roll
        Reset <= '0';
        if i >= 12 then Tnext <= 3;
        elsif Roll = '1' then
          Sum <= Sumarray(i);
          i := i + 1;
          Tnext <= 1;
        end if;
      when 1 => Rb <= '0'; Tnext <= 2;
      when 2 => Tnext <= 0;
        Trig1 <= not Trig1;                -- toggle Trig1
        if (Win or Lose) = '1' then
         Reset <= '1';
        end if;
      when 3 => null;                      -- Stop state
    end case;
  end process;

  process(CLK)
  begin
    if CLK = '1' and CLK'event then
      Tstate <= Tnext;
    end if;
  end process;
end dicetest;
```

图 5.18　掷骰子游戏测试平台的 VHDL 代码

Tester（图 5.19）把 DiceGame 模块和 GameTest 模块连接起来，这样我们就可以对整个游戏系统进行测试。图 5.20 给出了仿真器命令文件和输出。只要骰子滚动一次，此列表就被 Trig1 触发一次。命令“run 2000”使程序可以运行足够长的时间，这样就可以对所有的测试数据都进行操作。

```
entity tester is
end tester;

architecture test of tester is
component GameTest
  port(Rb, Reset: out bit;
       Sum: out integer range 2 to 12;
       CLK: inout bit;
       Roll, Win, Lose: in bit);
end component;
```

图 5.19　掷骰子游戏测试器

```
component DiceGame
  port(Rb, Reset, CLK: in bit;
       Sum: in integer range 2 to 12;
       Roll, Win, Lose: out bit);
end component;

signal rb1, reset1, clk1, roll1, win1, lose1: bit;
signal sum1: integer range 2 to 12;
begin
  Dice: Dicegame port map (rb1, reset1, clk1, sum1, roll1, win1, lose1);
  Dicetest: GameTest port map (rb1, reset1, sum1, clk1, roll1, win1, lose1);
end test;
```

图 5.19（续） 掷骰子游戏测试器

```
add list /dicetest/trig1 -NOTrigger sum1 win1 lose1 /dice/point
run 2000
```

ns	delta	trig1	sum1	win1	lose1	point
0	+0	0	2	0	0	2
100	+3	0	7	1	0	2
260	+3	0	11	1	0	2
420	+3	0	2	0	1	2
580	+2	1	4	0	0	4
740	+3	1	7	0	1	4
900	+2	0	5	0	0	5
1060	+2	1	6	0	0	5
1220	+3	1	7	0	1	5
1380	+2	0	6	0	0	6
1540	+2	1	8	0	0	6
1700	+2	0	9	0	0	6
1860	+3	0	6	1	0	6

图 5.20 掷骰子游戏测试器的仿真和命令文件

5.3 SM 图的实现

用来实现 SM 图的方法与实现状态图的方法是相似的。对于任何时序电路，我们均可以用一个组合电路和一些触发器（用于存储电路的状态信息）将其实现。在一些情况下，在 SM 图中可以确定等价的状态，也可以删除冗余的状态，其方法与化简状态表的方法一样。然而，因为在每个状态并不是对所有的输入都进行检测，所以 SM 图通常是不完整的，这就使化简过程更加困难。即使 SM 图中的状态数可以减少，但是通常我们并不希望这么做，因为状态的合并化简，使得 SM 图更加难以解读。

在通过 SM 图获得下一状态和输出方程之前，必须要先进行状态赋值。状态赋值的最好方式取决于 SM 图是如何实现的。如果使用门和触发器实现（或等价的 PLD 实现），则 1.7 节中介绍的状态赋值方法就可以用得上了。如果使用可编程门阵列实现，则可以使用 6.9 节中介绍的单热状态赋值方法。

下面，我们考虑如何实现图 5.21 给出的 SM 图。状态赋值如下：AB=00 代表 S_0，AB=01 代表 S_1，AB=11 代表 S_2。状态赋值后，就可以直接从 SM 图中得到输出方程和下一状态方程。因为 Moore 电路的输出 Z_a 只有在状态 00 时才为 1，所以 $Z_a = A'B'$。同样地，可以得到 $Z_b = A'B$，$Z_c = AB$。条件输出 $Z_1 = ABX'$，这是因为通过 Z_1 所在的链路以 $AB = 11$ 开始的，并且通过了 $X = 0$ 分支。同样地，可以得到 $Z_2 = ABX$。有三条链路（图 5.7(b)中的链路 1, 2, 3）终点为 $B = 1$ 状态。链路 1 从当前状态 $AB = 00$ 开始，经过分支 $X = 1$，并以 B=1 状态为终点。因此当 $A'B'X' = 1$ 时，B 的下一状态（B^+）为 1。链路 2 起始于状态 01，经过分支 $X = 1$，并以状态 11 为终点，因此 B^+ 方程中含有项 $A'BX$。同样地，从链路 3 中可以得到 B^+方程的另外一项 ABX。因此，B 的下一状态方程有三项，分别对应于三条链路：

$$B^+ = A'B'X + A'BX + ABX$$

链路 1　链路 2　链路 3

同样地，有两条链路的终点为 $A = 1$ 的状态，因此有

$$A^+ = A'BX + ABX$$

这些输出方程和下一状态方程可以通过卡诺图进行化简，将没有用到的状态 $AB = 10$ 作为随意项。

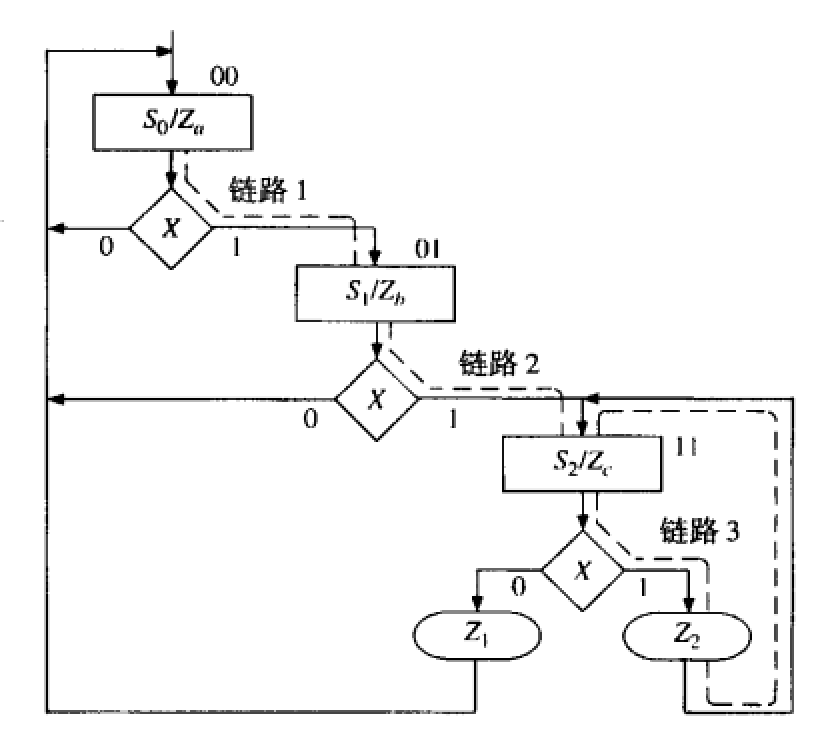


图 5.21　SM 图的实现

上面我们对触发器 A 和 B 进行了讨论。从 SM 图中得到触发器 Q 下一状态方程的推导步骤如下：

1. 确定所有 $Q = 1$ 的状态。
2. 对于每个状态，找出所有进入到该状态的链路。
3. 对于每个链路，沿着此条链路找到一个值为 1 的项。也就是说，从状态 S_i 到状态 S_j 的链路，如果状态机处于状态 S_i 并满足到达状态 S_j 的条件，则该项为 1。
4. 将步骤 3 中所有得到的项做或运算，就可以得到 Q^+（Q 的下一状态）的表达式。

5.3.1　二进制乘法器控制器的实现

下面再次观察乘法器控制器的 SM 图（图 5.22）。

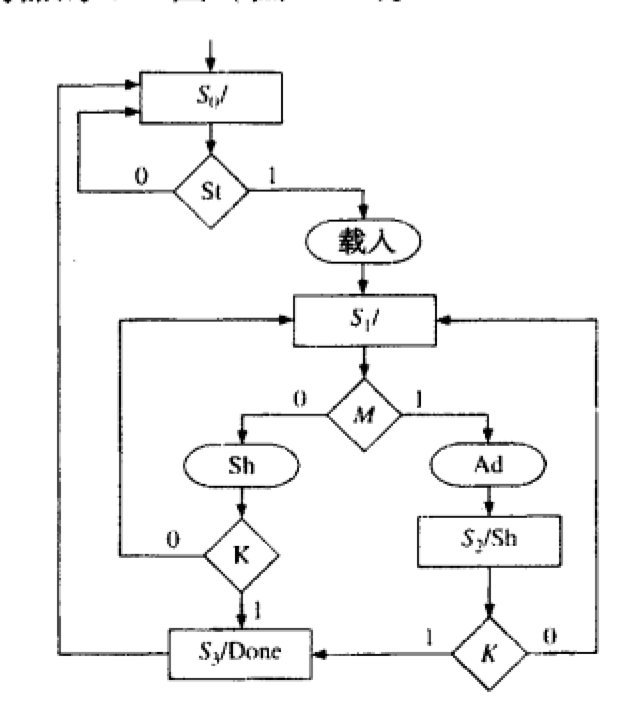


图 5.22　乘法器控制器的 SM 图

我们可以使用两个 D 触发器和一个组合逻辑电路实现此 SM 图。状态赋值为：$AB = 00$ 代表 S_0，$AB = 01$ 代表 S_1，$AB = 10$ 代表 S_2，$AB = 11$ 代表 S_3。

我们可以通过追溯 SM 图中的链路得到乘法器控制器的逻辑方程和下一状态方程，然后对得到的方程进行化简。首先考虑控制信号。只有在状态 S_0 下且 St 为真时，$Load$ 才为真。因此，$Load = S_0St = A'B'St$。同理，只有在状态 S_1 下且 M 为真时，Ad 才为真。因此，$Ad = A'BM$。在状态 S_3，*Moore* 电路输出 *Done* 信号，因此 $Done = S_3 = AB$。总之，乘法器控制器的逻辑方程为

$$Load = A'B'St$$
$$Sh = A'BM'(K'+K) + AB'(K'+K) = A'BM' + AB'$$
$$Ad = A'BM$$
$$Done = AB$$

下一状态表达式可以通过观察 SM 图和状态赋值得出。在状态 S_2 和 S_3，A 为真；若当前状态为 S_1 且 M 为真（$A'BM$），则下一状态为 S_2；若当前状态为 S_1，且 M 为假，K 为真（$A'BM'K$），则下一状态为 S_3；若当前状态为 S_2，且 K 为真（$A'BM'K$），则下一状态也为 S_3。因此，我们可以得到

$$A^+ = A'BM'K + A'BM + AB'K = A'B(M+K) + AB'K$$

同理，我们可以得到 B 的下一状态方程为

$$B^+ = A'B'St + A'BM'(K'+K) + AB'(K'+K) = A'B'St + A'BM' + AB'$$

此乘法器的控制器可以用两个触发器和几个逻辑门实现。逻辑门电路用于实现下一状态表达式和控制信号表达式。该电路可以由分离门电路来实现，或者用 PLA, CPLD 或 FPGA 来实现。

表 5.1 给出了乘法器控制器的状态转移表。表格中的每一行都与 SM 图中的一条链路相对应。因为状态 S_0 有两个输出通路，所以表中有两行对应于当前状态 S_0。第一行对应 $St = 0$ 输出通路，所以下一状态为 S_0，且输出也为 0。第二行对应 $St = 1$ 的输出通路，所以下一状态为 01 并且输出为 1000。由于在状态 S_1, S_2, S_3 中没有对 St 进行检测，所以在对应的行中就不用考虑 St。对于每一行的输出可以根据 SM 图中相应的链接路径来确定。例如，从 S_1 到 S_2 的链接路径，经过了条件输出 Ad，所以在这一行中 $Ad = 1$。由于 S_2 中含有 Moore 电路的输出 Sh，所以对应 $AB = 10$ 的两行中 $Sh = 1$。

表 5.1 乘法器控制器的状态转移表

	A	B	St	M	K	A⁺	B⁺	Load	Sh	Ad	Done
S_0	0	0	0	—	—	0	0	0	0	0	0
	0	0	1	—	—	0	1	1	0	0	0
S_1	0	1	—	0	0	0	1	0	1	0	0
	0	1	—	0	1	1	1	0	1	0	0
	0	1	—	1	—	1	0	0	0	1	0
S_2	1	0	—	—	0	0	1	0	1	0	0
	1	0	—	—	1	1	1	0	1	0	0
S_3	1	1	—	—	—	0	0	0	0	0	1

此控制器也可以使用 ROM 来实现。在用 ROM 进行实现时，首先我们要计算所需 ROM 的大小。此控制器的组合电路有 5 个不同的输入（A, B, St, M 和 K）。因此，ROM 必须有 32 个入口。由于组合电路应该生成 6 个信号（4 个控制信号，2 个下一状态信号），所以每个入口都应该为 6

位。因此，此设计可以用32×6的 ROM 和两个 D 触发器实现。如果我们用 PLA 实现组合逻辑电路，而不是用 ROM 来实现，那么 PLA 表与前面所说的状态转移表将完全相同。PLA 要有 5 个输入、6 个输出和 8 个乘积项。

如果使用 ROM，由于有 5 个输入，所以表格必须扩展到 $2^5=32$ 行。要扩展这个表格，就必须由所有的 0, 1 的可能的组合来代表表中每一行的短划线。如果某行中有 N 条小短划线，则此行就必须由 2^n 个行来代替。例如，表 5.1 中的第 5 行可以用下面的 4 行代替，其中新加入的入口用粗体字体表示：

0	1	**0**	1	**0**	1	0	0	0	1	0
0	1	**0**	1	**1**	1	0	0	0	1	0
0	1	**1**	1	**0**	1	0	0	0	1	0
0	1	**1**	1	**1**	1	0	0	0	1	0

5.4 掷骰子游戏的实现

如图 5.23 所示，掷骰子游戏（图 5.13）的 SM 图可以用一个组合电路和 3 个 D 触发器来实现。我们使用直接二进制状态赋值。组合电路有 9 个输入和 7 个输出，其中 3 个输入对应当前状态，3 个输出提供下一状态信息。所有的输入和输出均在表 5.2 的上部列出。状态转移表中的每一行都对应 SM 图中一条链路。在状态 $ABC=000$ 中，下一个状态 $A^+B^+C^+=000$ 或 001，这取决于 Rb 的值。由于状态 001 有 4 个输出通路，所以相应的表格中有 4 行。当 Rb 为 1 时，*Roll* 为 1 并且没有状态转换。当 $Rb=0$ 且 D_{711} 为 1 时，下一个状态为 010。当 $Rb=0$ 且 $D_{2312}=1$ 时，下一个状态为 011。在从状态 001 到 100 的链路上，Rb, D_{711}, D_{2312} 都为 0，*Sp* 为条件输出。这个链路对应于表格中的第 4 行，在这行中有 $Sp=1$ 和 $A^+B^+C^+=100$。在状态 010，信号 *Win* 总是开启的，下一状态为 010 或 000，这取决于 *Reset* 信号的值。同理，在状态 011 中，*Lose* 信号也总是开启的。在状态 101 中，如果 $Eq=1$，则 $A^+B^+C^+=010$，否则 $A^+B^+C^+=011$ 或 100，这取决于 D_7 的值。因为没有使用状态 110 和 111，所以状态 $ABC=110$ 和 111 的下一状态和输出均为随意项。

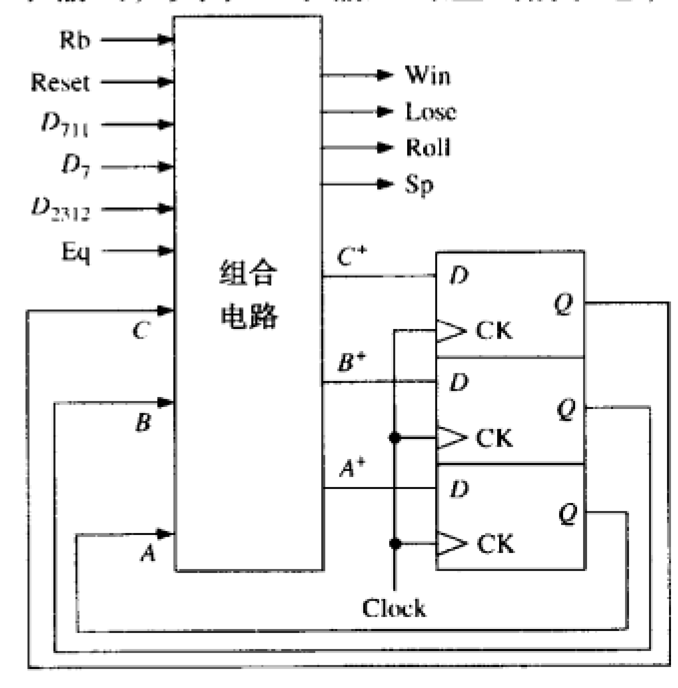


图 5.23 掷骰子游戏控制器的实现

我们可以从表 5.2 中推导得出控制信号和下一状态表达式。我们可以利用嵌入变量的卡诺图法（见第 1 章）或者使用 CAD 辅助程序例如 LogicAid，从表 5.2 中得到想要的表达式。同样，我们也可以通过追踪 SM 图中的各条链路得到想要的表达式，然后通过使用下一状态中的随意项对表达式进行化简。

图 5.24 给出了 A^+, B^+和 *Win* 的卡诺图。由于 A, B, C 和 Rb 在表格的大多数行中都已经被赋值，所以使用这四个变量就用在卡诺图的外边，剩余的变量则被放在卡诺图里。图中的 E_1, E_2, E_3 和 E_4 所代表的表达式见图下注释。观察状态转移表中的 A^+列，在第 4 行 A^+为 1，所以在卡诺图中 $ABCRb=0010$ 的方格中，我们应填入 $D'_{711}D'_{2312}$。为了节省空间，我们定义 $E_1=D'_{711}D'_{2312}$，并在方格中填入 E_1。由于在 11, 12, 16 行中 A^+为 1，因此在 1000, 1001, 1011 的方格中填入 1。根据 13

行中的数据，我们将 $E_2 = D'_7Eq'$填入 $ABCRb = 1010$ 所对应的方格中。在第 7, 8 行，由于当 $ABC = 010$ 时 Win 均为 1，所以我们在输出为 Win 的卡诺图的相应方格中填入 1。

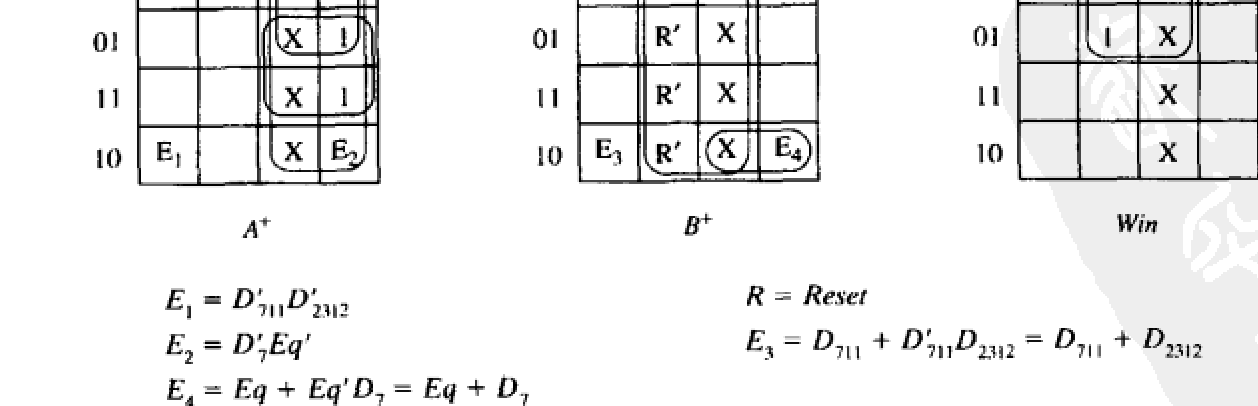
最终的表达式为

$$
\begin{aligned}
A^+ &= A'B'C\,Rb'D'_{711}D'_{2312} + AC' + ARb + AD'_7Eq' \\
B^+ &= A'B'C\,Rb'(D_{711} + D_{2312}) + B\,Reset' + AC\,Rb'(Eq + D_7) \\
C^+ &= B'Rb + A'B'C\,D'_{711}D_{2312} + BC\,Reset' + AC\,D_7Eq' \\
Win &= BC' \\
Lose &= BC \\
Roll &= B'C\,Rb \\
Sp &= A'B'C\,Rb\,D'_{711}D'_{2312}
\end{aligned}
\tag{5.1}
$$

这些表达式可以使用任何标准技术实现（如分立门电路、PAL、GAL、CPLD 或 FPGA）。

表 5.2 骰子游戏状态转移表（PLS 表）

	ABC	Rb	Reset	D_7	D_{711}	D_{2312}	Eq	A^+	B^+	C^+	Win	Lose	Roll	Sp
1	000	0	—	—	—	—	—	0	0	0	0	0	0	0
2	000	1	—	—	—	—	—	0	0	1	0	0	0	0
3	001	1	—	—	—	—	—	0	0	1	0	0	1	0
4	001	0	—	—	0	0	—	1	0	0	0	0	0	1
5	001	0	—	—	0	1	—	0	1	1	0	0	0	0
6	001	0	—	—	1	—	—	0	1	0	0	0	0	0
7	010	—	0	—	—	—	—	0	1	0	1	0	0	0
8	010	—	1	—	—	—	—	0	0	0	1	0	0	0
9	011	—	1	—	—	—	—	0	0	0	0	1	0	0
10	011	—	0	—	—	—	—	0	1	1	0	1	0	0
11	100	0	—	—	—	—	—	1	0	0	0	0	0	0
12	100	1	—	—	—	—	—	1	0	1	0	0	0	0
13	101	0	—	0	—	—	0	1	0	0	0	0	0	0
14	101	0	—	1	—	—	0	0	1	1	0	0	0	0
15	101	0	—	—	—	—	1	0	1	0	0	0	0	0
16	101	1	—	—	—	—	—	1	0	1	0	0	1	0
17	110	—	—	—	—	—	—	—	—	—	—	—	—	—
18	111	—	—	—	—	—	—	—	—	—	—	—	—	—



$E_1 = D'_{711}D'_{2312}$

$E_2 = D'_7Eq'$

$E_4 = Eq + Eq'D_7 = Eq + D_7$

$R = Reset$

$E_3 = D_{711} + D'_{711}D_{2312} = D_{711} + D_{2312}$

图 5.24 从表 5.2 推导得到的图

如果用 ROM 实现掷骰子游戏的控制器，则必须有 512 个入口地址（因为有 9 个输入），且每

个入口单元都是 7 位的（3 位用于下一状态，4 位用于输出）。由于输入量很大，所以就需要一个很大的 ROM。因此，对于输入量很大的状态机来说，使用 ROM 对其进行实现并不是一个很好的选择。

下面，我们根据图 5.11 的框图和式(5.1)，开始编写掷骰子游戏控制器的数据流描述方式 VHDL 模块，如图 5.25 所示。在时钟上升沿到来时，进程更新触发器的状态寄存器和指针寄存器。控制信号和 *D* 触发器的输入等式都由并发语句实现。特别地，D_7, D_{711}, D_{2312} 和 *Eq* 是通过条件赋值语句实现的。另外，还有一种实现方法：所有信号和 *D* 触发器输入表达式均在进程中实现，且此进程的敏感变量为 *A*, *B*, *C*, *Sum*, *Point*, *Rb*, D_7, D_{711}, D_{2312}, *Eq* 和 *Reset*。如果把图 5.25 和图 5.19 的测试模块联合使用，则所得到的结果与图 5.15 所示的行为描述模块的结果是相同的。

```
architecture Dice_Eq of DiceGame is
signal Sp,Eq,D7,D711,D2312: bit:= '0';
signal DA,DB,DC,A,B,C: bit:='0';
signal Point: integer range 2 to 12;
begin
  process(CLK)
  begin
    if CLK = '1' and CLK'event then
      A <= DA; B <= DB; C <= DC;
      if Sp = '1' then Point <= Sum; end if;
    end if;
  end process;
  Win <= B and not C;
  Lose <= B and C;
  Roll <= not B and C and Rb;
  Sp <= not A and not B and C and not Rb and not D711 and not D2312;
  D7 <= '1' when Sum = 7 else '0';
  D711 <= '1' when (Sum = 11) or (Sum = 7) else '0';
  D2312 <= '1' when (Sum = 2) or (Sum = 3) or (Sum = 12) else '0';
  Eq <= '1' when Point = Sum else '0';
  DA <= (not A and not B and C and not Rb and not D711 and not D2312) or
        (A and not C) or (A and Rb) or (A and not D7 and not Eq);
  DB <= ((not A and not B and C and not Rb) and (D711 or D2312)) or
        (B and not Reset) or ((A and C and not Rb) and (Eq or D7));
  DC <= (not B and Rb) or (not A and not B and C and not D711 and D2312) or
        (B and C and not Reset) or (A and C and D7 and not Eq);
end Dice_Eq;
```

图 5.25　掷骰子游戏的数据流描述模块

为了完成掷骰子游戏的 VHDL 程序，我们添加了两个模 6 计数器，如图 5.26 和图 5.27 所示。计数器的初值为 1，所以两个骰子的和数将会一直处在 2 ~ 12 之间。当 *Cnt*1 在状态 6 时，下一个时钟的到来使其变为状态 1，并且 *Cnt*2 加 1（如果在状态 6，则 *Cnt*2 置 1）。

```
entity Counter is
  port(Clk, Roll: in bit;
       Sum: out integer range 2 to 12);
end Counter;

architecture Count of Counter is
signal Cnt1, Cnt2: integer range 1 to 6:= 1;
begin
  process(Clk)
  begin
    if Clk = '1' then
      if Roll = '1' then
        if Cnt1 = 6 then Cnt1 <= 1; else Cnt1 <= Cnt1 + 1; end if;
        if Cnt1 = 6 then
          if Cnt2 = 6 then Cnt2 <= 1; else Cnt2 <= Cnt2 + 1; end if;
        end if;
      end if;
    end if;
  end process;
  Sum <= Cnt1 + Cnt2;
end Count;
```

图 5.26　掷骰子游戏中的计数器模块

本节介绍了 SM 图的实现方法。我们可以使用分立门电路、PLA、ROM 或 PAL 实现 SM 图。我们可以通过在电路中加入其他元件，来减小 PLA 或 ROM 的大小。这些方法都是把 SM 图变为各种不同形式和技术来实现设计的。我们还可以用微程序来实现 SM 图。

```
entity Game is
  port(Rb, Reset, Clk: in bit;
       Win, Lose: out bit);
end Game;

architecture Play1 of Game is
component Counter
  port(Clk, Roll: in bit;
       Sum: out integer range 2 to 12);
end component;

component DiceGame
  port(Rb, Reset, CLK: in bit;
       Sum: in integer range 2 to 12;
       Roll, Win, Lose: out bit);
end component;

signal roll1: bit;
signal sum1: integer range 2 to 12;
begin
  Dice: Dicegame port map (Rb, Reset, Clk, sum1, roll1, Win, Lose);
  Count: Counter port map (Clk, roll1, sum1);
end Play1;
```

图 5.27 完整的掷骰子游戏模块

5.5 微程序

微程序是一种实现数字系统控制单元的技术。为了实现控制单元，我们可以绘制状态图或 SM 图，写出控制器输出和下一状态的逻辑表达式，并使用门电路和触发器实现状态机。在 5.3 节和 5.4 节中，我们用以上步骤分别实现了二进制乘法器和掷骰子游戏。这种实现方法称为硬线化（hardwiring）技术，因为表示控制信号是由固定的（硬线的）逻辑电路实现的。

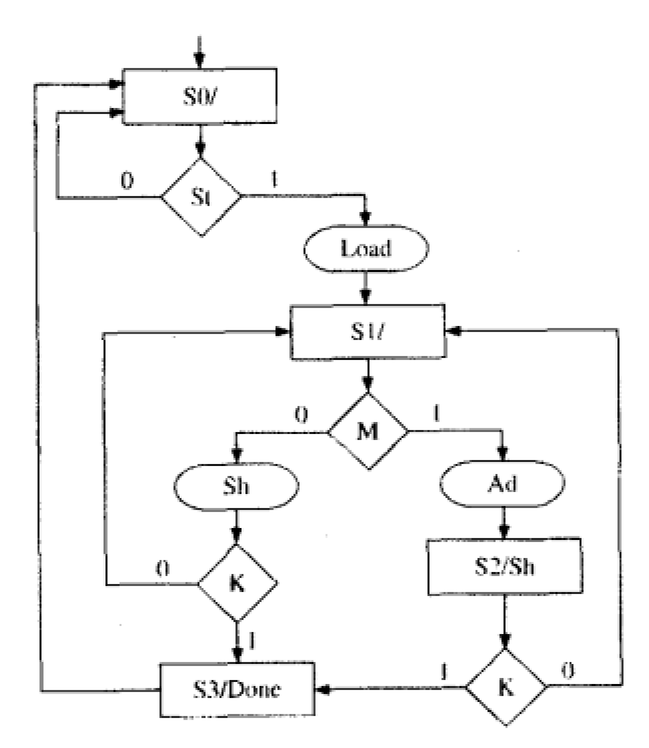
相对地，还有另外一种实现复杂数字系统控制单元的方法：称为微程序。它是由 Maurice Wilkes 在 1951 年提出的。微程序可以构建一台特殊的计算机，该计算机可以执行描述系统控制器的算法流程图。微程序技术源于结构（architecturre）和控制器的分离（在第 4 章的开头已经加以介绍）。一旦结构和控制器确定，则控制器的流程图就可以系统地规定从复位状态到其他所有状态的控制流程中各个时刻应该产生的所有控制信号。观察二进制移位-相加乘法器的 SM 图 [图 5.28(a)]。我们可以写出该乘法器控制操作的伪码，如图 5.28(b)所示。该乘法器在第 4 章中已经详细介绍过了。

对控制器的这种描述，容易使我们把控制器的运作同普通计算机的程序对应起来。微程序的开发正是从这一实现开始的。

如果一个存储器能够存储所有的控制信号和每个状态和每一输入条件下的下一状态信息，则我们仅通过对存储器中的内容“排队”就可以实现控制器。正因如此，基于微程序的控制器通常称为排序器。存储控制字的内存称为控制存储器或微程序存储器。

当数字系统的复杂度太高的时候，微程序技术就非常有魅力。由于过去我们都是通过人工进行调试的，所以很难发现和纠正错误。微程序的系统特性使系统调试变得简单。系统的更改相对容易实现。错误可以定位和容易纠正。正是这些特性使微程序非常流行。

微程序的缺点是它的速度慢。这是因为其存储器访问是通过读取控制存储器中的控制字来实现的。而硬线化控制信号是由逻辑门生成的，而且逻辑门通常比存储器要快，所以用硬线化技术设计的系统速度较快。



(a) 乘法器控制器的 SM 图

```
S0:  if St is true, produce Load Signal and go to S1,
           else return to S0
S1:  if M is true, produce Ad and go to S2,
           else produce Sh, check whether K is 1;
           if K is 1 go to S3;
           if K is 0, go to S1;
S2:  produce Sh;
           if K = 0, go to S1;
           else go to S3;
S3:  produce Done and go to S0
```

(b) 乘法器控制器操作伪码

图 5.28　乘法器控制器的 SM 图和操作流程

早期的微处理器都是微程序化的，如 Intel 8086 和 Motorola 68000，这些微处理器都支持多种基址和变址寻址方式，允许直接从内存中读操作数和写结果，还有很多复杂指令用以执行一系列的基本操作。一条复杂指令涉及好几个操作，而所需要的控制信号可以系统地指定在微程序字中，因此微程序就显得很方便。如果采用硬线化技术，就很难实现这些微处理器了。

随后很多东西都发生了变化。在 20 世纪 70 年代，大家发现许多微处理器一半以上的芯片面积都用于实现控制器（比如，处理器的数据通道在芯片上所占面积小于一半）。微处理器的复杂性使得研究人员和设计者不得不寻求更简单的实现方法，这就使大家都进入了 RISC（Rededed Instruction Set Computing，精简指令集计算）时代。RISC 微处理器更加简单，包含较少的内存寻址方式，并且需要简单的控制单元。计算机辅助设计（CAD）工具的改进提高了设计者的调试能力。现在，微程序只用于使用复杂指令集结构（ISA）的微处理器中，但是它仍不失为一个非常重要的概念和一流的方法。

微程序可以通过很多方法来实现。一般的思路是存储每一个状态所对应的控制字，这一控制字也称为微指令。微指令规定将要生成的输出，同时也指定下一条微指令的位置，这与状态图或 SM 图中的状态转移相对应的。

5.5.1 双地址微代码

图 5.29 给出了一个典型的微程序实现的硬件布局。每个 ROM 单元都存储一个控制字或微指令，状态寄存器 ROM 的唯一输入。一个多路选择器用于每个输入的选择性测试，在每个状态最多选择一个变量进行测试，指出所选的控制信号的真伪（用 TEST 选择）。另一个多路选择器则用于选择控制转移的下一状态。这种技术称为双地址微代码，因为与测试信号的真伪条件相对应的下一两个状态是由微指令明确指定的。

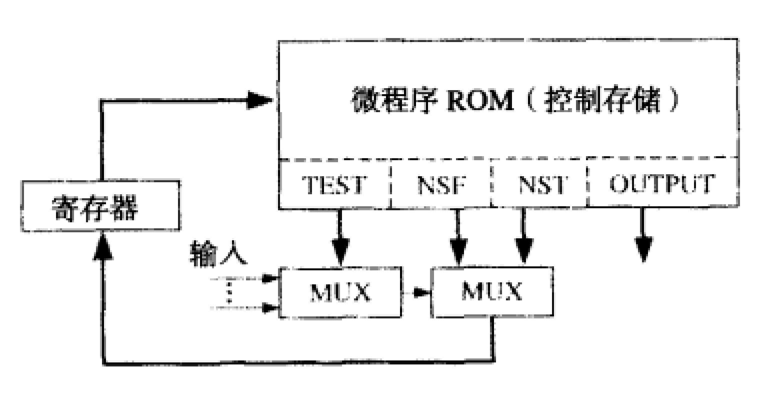


图 5.29 微程序的典型硬件布置

ROM 输出有四个区域：TEST, NSF, NST 和 OUTPUT。TEST 控制第一个 MUX 的输入，选择在每个状态下要检测的输入。如果这一输入为 0（逻辑伪），则第二个 MUX 选择 NSF 作为下一状态。如果该输入为 1（逻辑真），则第二个 MUX 选择 NST 作为下一状态。OUTPUT 域的每位比特与控制信号相对应。注意，为了能够使用该硬件布局，SM 图必须只有 Moore 输出，因为该输出只能是状态的函数。

针对微程序的 SM 图变换

为了简便有效地实现微程序编程，我们要变换 SM 图。我们不想使用简单的查找表方法（在 LUT 法中直接规定了所有输入组合和当前状态）。我们对 SM 图进行变换，使每个状态只有一个入口。这种变换可能会增加状态数，但是最终得到的微程序的大小仍旧比基于 LUT 法的 ROM 的大小要小得多。

去除条件输出

我们希望构建 Moore 控制器，这样就没有条件控制信号。如果控制信号依赖于某些输入条件，则我们应该把不同输入组合所对应的控制信号都存起来。因此，为了便于实现微程序，我们首先把状态图或 SM 图转换为 Moore 状态机。通过添加适当数量的状态，我们也可以把任何 Mealy 机转换为 Moore 机。

每个状态只允许一个限制量

在微程序中关联文献中，状态机的每个状态中均要被检测的输入称为限制量（qualifier）。例如，图 5.28 中的 *St*, *M* 和 *K* 均为限制量。图中 S_0 和 S_2 都只有一个限制量，但是状态 S_1 检测了两个限制量 *M* 和 *K*。如果图 5.28 使用伪码编写，则需要对 S_1 中的多个限制量使用 if 嵌套语句。当每个状态含有多个限制量时，微程序也可以进行实现。但是当每个状态只有一个限制量时，微程序的实现会简单得多。

这样，如果对 SM 图做如下两个变换，则微程序会变得简单得多。

1. 变换 SM 图为 Moore 机，去除 SM 图中所有的条件输出。
2. 每个状态只检测一个输入（限制量）。

下面，我们把乘法器的 SM 图变换为便于微程序的形式。首先，通过对每个条件输出（例如 SM 图中的每个椭圆框）添加状态把 SM 图转换为 Moore 机。因此，为了表达条件输出 *Load*，我们在状态 S_0 后添加状态 S_{01}；为了表达输出 *Ad*，我们在状态 S_1 后添加状态 S_{11}；为了条件输出 *Sh*，我们在状态 S_1 后添加状态 S_{12}。这样，在任何状态下，我们都只需检测一个限制量。更改后的 SM 图见图 5.30。

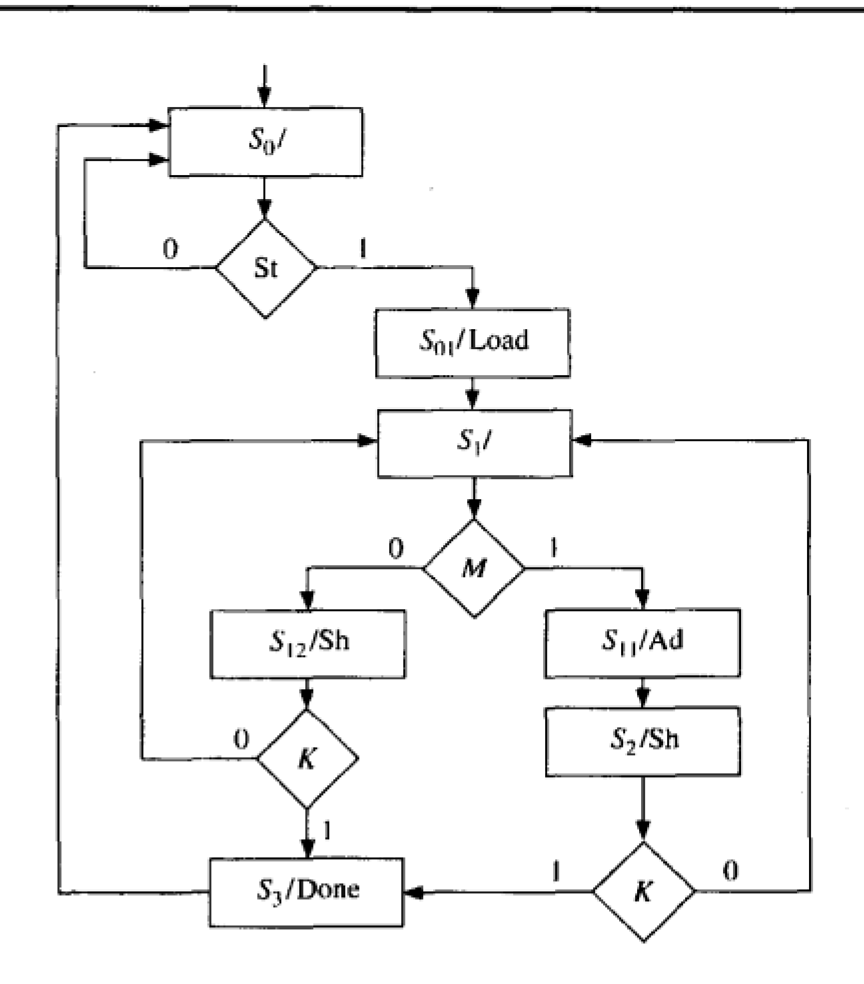


图 5.30　无条件输出的乘法器 SM 图（从图 5.28 导出）

我们使用下列伪码描述相应的操作：

```
S0:  if St is true, go to S01,
             else go to S0;
S01: produce Load; Go to S1;
S1:  if M is true, go to S11, else go to S12;
S11: produce Ad; go to S2;
S12: produce Sh; if K = 0, go to S1; else go to S3;
S2:  produce Sh;
             if K=0, go to S1;
             else go to S3;
S3:  produce Done; go to S0;
```

这时候要检查变换后的 SM 图，并去掉多余的状态。状态 S_{11} 和 S_2 可以合并吗？由于在移位前要进行相加操作，所以控制信号 *Ad* 应该在控制信号 *Sh* 之前出现。因此，状态 S_{11} 和 S_2 不可以合并。

下面我们检查状态 S_{12} 和 S_2。状态 S_{12} 和 S_2 所执行的操作完全相同，并且下一状态也相同，所以可以把这两个状态合并。这就是一个转换后潜在状态最小化的例子。设合并后的状态为 S_2。改进后的 SM 图如图 5.31 所示。

假设按照 $S_0, S_{01}, S_1, S_{11}, S_2, S_3$ 的顺序直接进行二进制状态赋值，则微程序如表 5.3 所示。由于只有三个输入（*St, M* 和 *K*），所以一个 4 选 1 MUX 就足以完成限定量的正确选择。该 MUX 的连接如图 5.32 所示。

观察表 5.3 中的第一行，它对应于状态 S_0（编码为 000）。要检测的输入为 *St*。由于 *St* 与多路选择器的输入 0 相连，所以这一行的 TEST 域为 00。如果 *St* 为假，则下一状态为 S_0，且 NSF 域为 000。如果 *St* 为真，则下一状态为 S_{01}，且 NSF 域为 001。在状态 S_0 中，控制信号 *Load, Ad, Sh, Done* 均为 0。

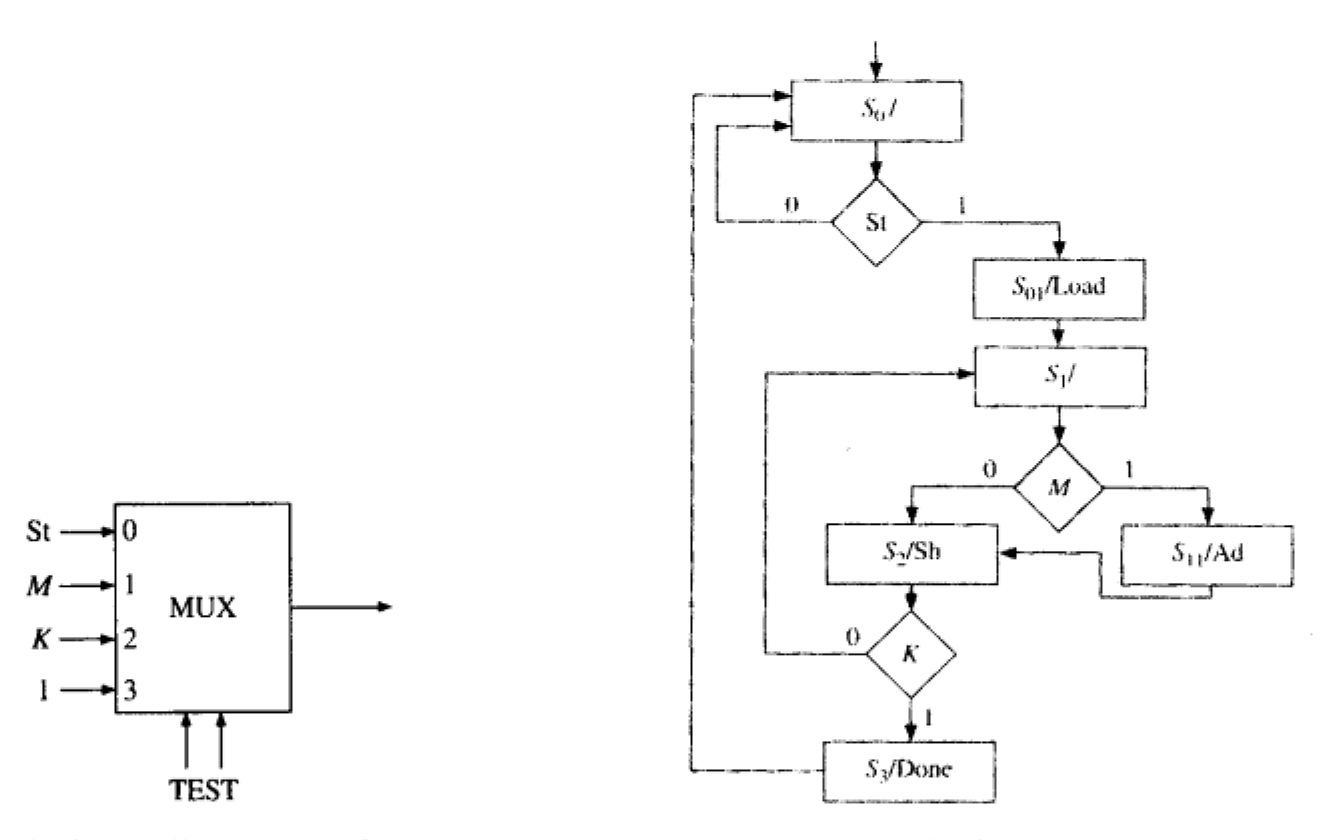


图 5.31　对图 5.30 进行状态最小化后得到的乘法器的 SM 图　图 5.32　双地址微程序乘法器中的 4 选 1 MUX

表 5.3　乘法器的双地址微程序，NST 和 NSF 均具体设定（对应于图 5.29）

状　态	ABC	TEST	NSF	NST	Load	Ad	Sh	Done
S_0	000	00	000	001	0	0	0	0
S_{01}	001	11	010	010	1	0	0	0
S_1	010	01	100	011	0	0	0	0
S_{11}	011	11	100	100	0	1	0	0
S_2	100	10	010	101	0	0	1	0
S_3	101	11	000	000	0	0	0	1

状态 S_{01} 的微代码在表 5.3 的第二行给出。状态 S_{01} 生成 *Load* 信号，并且控制器状态转移到 S_1，不检测任何输入信号。我们把图 5.32 中多路选择器中的最后一个未使用的输入管脚赋值为'1'。对应于多路选择器的最后一个输入，我们把 TEST 域记为 11。在状态 S_1 中，对输入信号 *M* 进行检测。由于 *M* 与多路选择器的输入 1 相连，所以第三行中 TEST 域为 01。根据相同的规则，我们可以把表 5.3 的所有行填满。

由于有 6 个状态，所以需要使用 3 个触发器，存储该微程序的 ROM 要有 6 个入口，对应于每个状态，且每个存储单元均需 12 位，其中 2 位用于 TEST，3 位用于 NST，4 位用于控制信号 *Load, Ad, Sh* 和 *Done*。*ABC* 用来表示存储微指令的地址。

图 5.29 给出了一个微程序的硬件布局，此微程序具有两个下一状态地址且每个状态均只有一个限制量。单限制量微程序是指在某个状态中只有一个输入被检测。双地址微代码是指根据两个可能输入值在控制字中明确指定下一状态（如果输入为真，则下一状态为 NST；反之 NSF）(图 5.29 可以修改为允许 Mealy 输出，只要把 OUTPUT 域用 OUTPUTF 或 OUTPUTT 取代，并用一个 MUX 选择这两个输出域就可以)。

5.5.2　单限制量、单地址微代码

在表 5.3 的微程序中，每个微指令都可以指定两个可能的下一状态——当输入为真时的下一状态和当输入为假时的下一状态。不同状态的微代码可以按任意顺序排列，因为每个状态的下一

条微指令是确定的，没有任何默认的控制流程。

前面谈到的微程序类似于软件，但是传统的程序中，除了转移指令和跳转指令可以改变控制流外，控制流程是顺序的。如果不进入一个分支，则控制流程就进到下一条指令。基于这种类似的结构，每个微程序完全只需指定一个下一状态地址即可。

下面我们考虑应如何做才能使下一行的状态成为默认的下一状态。为此，状态的赋值应该做到：如果限制量（输入）为假，则下一状态值应为当前状态值加 1。如果限制量为真，则在微代码中明确指定的状态为唯一的下一状态。如果限制量为假，则控制简单地移到下一行获取下一条微指令。

这种微程序可以用图 5.33 给出的硬件布局来实现。由于控制流程通常只是向前走一步到达下一个位置，所以用一个计数器足够。该计数器类似于一个微处理器的程序计数器（PC）。计数器指向控制器的当前状态，类似于 PC 指向下一条要获取的指令。每个 ROM 单元都存储一个控制字或微指令。OUTPUT 位与控制信号相对应。TEST 位指定要检测的限制量。NST 位给出当限制量为真时的目标微指令。一个多路选择器给出所选控制信号（由 TEST 给出）的真伪。当限制量为假时，计数器加 1 并指向下一条微指令，对应于默认的下一状态。当限制量为真时，计数器载入 NST 位并得到下一条微指令的存储器位置，这就是明确指定的下一状态。可并行载入的计数器是实现这一模块的最理想元件。多路选择器选出相关的限制量，其输出决定计数器是顺序计数，还是载入 NST 所指定的下一状态。

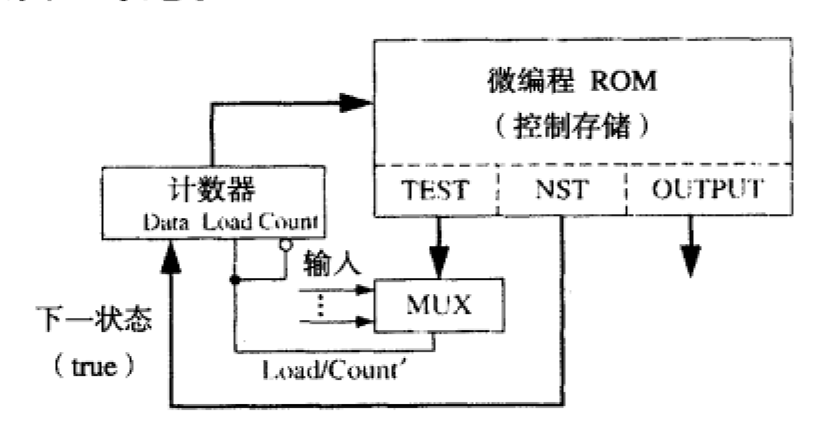


图 5.33 具有单地址微代码的微程序系统

单地址微代码的状态赋值必须十分小心（相反，在前面讨论的双地址微代码中，可以选用任意的状态赋值方法）。但地址微代码的状态赋值必须满足条件：对于每个状态，应该有一个下一状态，其状态赋值等于当前状态赋值的加 1（默认下一状态）。对于每个条件框来说，如果可能的话，每个伪分支的下一状态赋值应该是顺序递增的。如果做不到这样，则必须再添加额外状态（称为 *X* 状态）。对一长串的状态按顺序进行赋值可以减少所需 *X* 状态的个数，为此我们需要对一些检测的变量取补。

图 5.34 给出了改动后的二进制乘法器的 SM 图，针对单地址微代码，采用了顺序递增的状态赋值。在状态 *S*0，输入 *St* 取补，所以状态 *S*01 为默认的下一状态，如图 5.34(a)所示。如果输入 *St* 没有取补，则需要添加状态，如图 5.34(b)所示。状态 *S*2 是状态 *S*1 的默认下一状态。在状态 *S*2，我们使用 *K'*，因此状态 *S*2 的默认下一状态为 *S*3。这样，图 5.34(a)中，状态 *S*0, *S*01, *S*1, *S*2, *S*3 可以依次赋值为 0～4。当限制量为真时，下一状态可以赋任何值。状态 *S*11 赋值为 5。如果用变量 *K* 代替 *K'*，则当 *K* 为 0 时，需要在 *S*2 到 *S*1 的路径上添加状态。如图 5.34(b)所示，当输入变量不能取补时，需要添加两个状态。

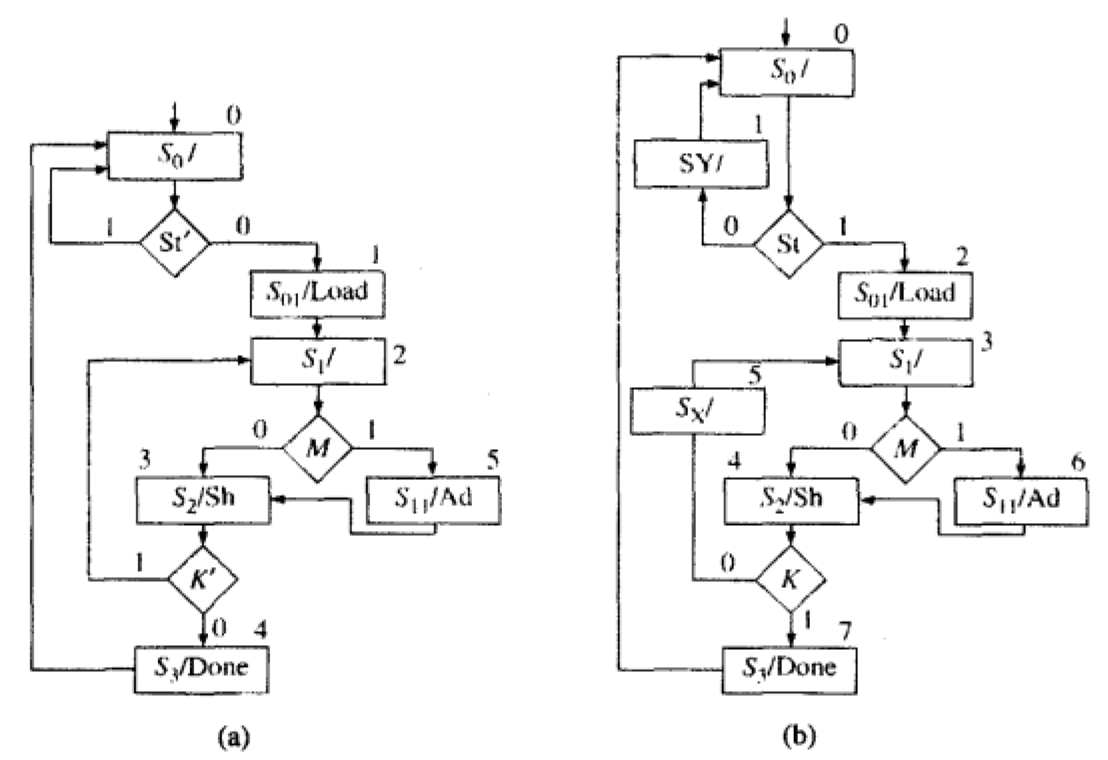


图 5.34 改动后的单地址微代码连续状态赋值的二进制乘法器 SM 图

表 5.4 给出了乘法器的单地址微程序。我们使用了改动的具有最少状态数的 SM 图[图 5.34(a)]。由于有 3 个输入：St', M 和 K'，所以使用一个 4 选 1 MUX 就可以选出正确的限制量。多路选择器的连接如图 5.35 所示。

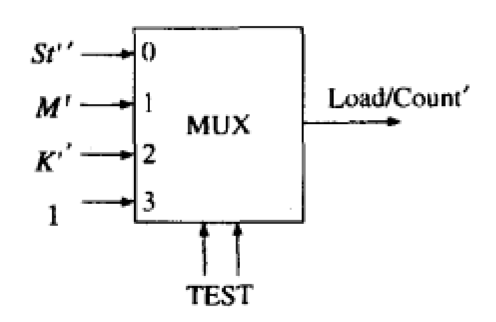


图 5.35 用微程序实现的乘法器中的多路选择器（单地址微代码）

表 5.4 给出的单地址微程序具有 6 个入口，且每个单元有 9 位。相反，表 5.3 给出的双地址微程序需要 6 个入口，且每个单元具有 12 位。

表 5.4 乘法器的单地址微程序（只对 NST 具体设定）

状 态	ABC	TEST	NST	Load	Ad	Sh	Done
S_0	000	00	000	0	0	0	0
S_{01}	001	11	010	1	0	0	0
S_1	010	01	101	0	0	0	0
S_2	011	10	010	0	0	1	0
S_3	100	11	000	0	0	0	1
S_{11}	101	11	011	0	1	0	0

如果采用常规的 LUT（ROM）法来实现该乘法器的控制器，则所需 ROM 的大小为32×6。由于有 4 个状态，所以需要 2 个触发器和 2 个下一状态表达式。由于还有 3 个输入：St, M 和 K。因此状态机的状态表有 32 行。由于有 2 个下一状态表达式和 4 个输出，所以每个入口地址为 6 位长。表 5.5 给出了查表法（ROM）和微代码法的比较。如果状态机输入变量数很大，则基于常规 LUT 法的 ROM 大小太大，很难实现。

表 5.5 乘法器控制器的不同实现方法比较

方 法	ROM 大小	
	入口数×宽度	位 数
基于原 SM 图的 LUT 法	32×6	192
双地址微代码	6×12	72
单地址微代码	6×9	54

5.5.3 掷骰子游戏控制器的微程序实现

下面我们使用前面介绍的微程序实现掷骰子游戏的控制器。它既可以使用单地址微代码实现，也可以使用双地址微代码实现。

掷骰子游戏控制器的双地址微代码实现

首先讨论掷骰子游戏控制器的双地址微代码实现，我们使用图 5.29 的硬件布局。为了使用微代码，我们需要进行 SM 图转换。首先，所以的输出必须转换为 Moore 输出；接着，在每个状态只能有一个输入变量被检测。这与图 5.29 中的方框图是直接对应的，因为在每个状态 TEST 区域只能选出一个输入进行检测，且输出仅与此状态决有关。图 5.36 给出了变换后的掷骰子游戏的 SM 图。

下面，我们采用直接二进制状态赋值给出微程序表（表 5.6）。由于必须对变量 Rb, D_{711}, D_{2312}, Eq, D_7 和 $Reset$ 进行检测，所以我们使用一个 8 选 1 MUX（图 5.37）。当 $TEST = 001$ 时，选中 Rb，依此类推。在状态 S_{13} 中，下一状态总是 0111，所以 $NSF = NST = 0111$，并且 TEST 域均为"随意项"。ROM 表中的每一行都与 SM 图中的一条链路相对应。例如，在状态 S_2 中，TEST 域为 110，则 $Reset$ 被选中。如果 $Reset = 0$，则选择 $NSF = 0100$；如果 $Reset = 1$，则选择 $NST = 0000$。在状态 S_2 中，输出 $Win = 1$ 并且其他输出为 0。

表 5.6 掷骰子游戏的双地址微程序

状 态	ABCD	TEST	NSF	NST	Roll	Sp	Win	Lose
S_0	0000	001	0000	0001	0	0	0	0
S_1	0001	001	0010	0001	1	0	0	0
S_{11}	0010	010	0011	0100	0	0	0	0
S_{12}	0011	011	0101	0110	0	0	0	0
S_2	0100	110	0100	0000	0	0	1	0
S_{13}	0101	xxx	0111	0111	0	1	0	0
S_3	0110	110	0110	0000	0	0	0	1
S_4	0111	001	0111	1000	0	0	0	0
S_5	1000	001	1001	1000	1	0	0	0
S_{51}	1001	100	1010	0100	0	0	0	0
S_{52}	1010	101	0111	0110	0	0	0	0

掷骰子游戏控制器的单地址微代码实现

单地址微代码使用的硬件框图见图 5.33。该电路中使用了一个计数器，而没有状态寄存器。只有一个目标，即 NST 域，要指定。TEST 域选择每个状态下的一个输入进行测试。如果该输入为 1（真），则 NST 域被载入到计数器。如果该输入为 0，则计数器加 1。

这种方法需要对 SM 图进行修改，如图 5.38 所示，并使用顺序状态赋值。如果不能按顺序进行状态赋值，则要添加额外的状态（称为 X 状态）。对一长串的状态按顺序进行赋值可以减少所需 X 状态的个数。为了将其实现，我们需要对一些被测试的变量取补。在图 5.38 中 Rb 和 $Reset$ 分别在两处被取补，并且 0, 1 分支也做出相应的改变。做出这些改变后，状态 0000, 0001, …, 1000

就是顺序赋值了。S_3赋值为 1001。在添加 X 状态之前，NSF 为 0000，NST 为 1001，所以两个下一状态中的任何一个都不是顺序的。因此我们加入一个 X 状态 S_X，并进行顺序赋值为 1010。S_X的下一个状态永远为 0000。如果我们把 S_2 赋值为 1011，而下一状态将为 1011 和 0000，则二者均不是顺序赋值的。我们可以通过添加一个 X 状态来解决这个问题。还有一个更好的方法就是把 S_2 赋值为 1111。由于 1111 加 1 后即为 0000，所以剩下的状态都是顺序赋值的，就不需要添加 X 状态了。

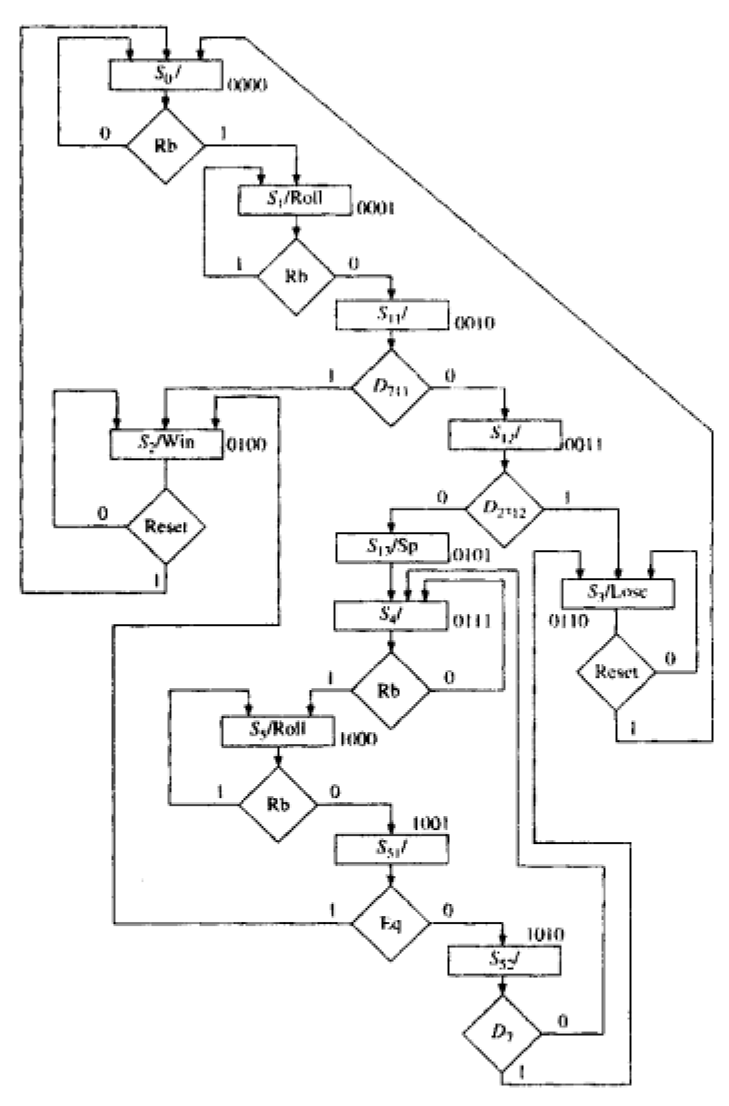


图 5.36 具有 Moore 输出和单限制量的 SM 图

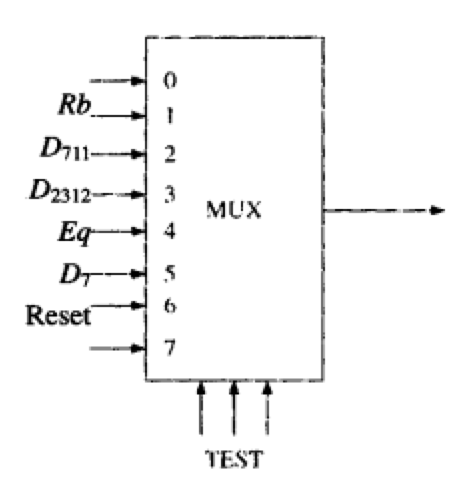


图 5.37 掷骰子游戏的双地址微代码实现中的 MUX

在图 5.39 中由 MUX 检测的输入与图 5.37 中检测的输入很相似，只是检测量 D_7 和 *Reset* 取补了，而且需要对 *Rb* 和 *Rb*′都进行检测。由于在状态 S_X 中 NST 永远为 0000，所以多路选择器的输入中有一个必须为 1。相应的微程序 ROM 表见表 5.7。

表 5.7 单地址微代码实现的掷骰子游戏的微程序

状 态	ABCD	TEST	NST	ROLL	Sp	Win	Lose
S_0	0000	000	0000	0	0	0	0
S_1	0001	001	0001	1	0	0	0
S_{11}	0010	010	1111	0	0	0	0
S_{12}	0011	011	1001	0	0	0	0
S_{13}	0100	111	0101	0	1	0	0
S_4	0101	000	0101	0	0	0	0
S_5	0110	002	0110	1	0	0	0
S_{51}	0111	100	1111	0	0	0	0
S_{52}	1000	101	0101	0	0	0	0
S_3	1001	110	1001	0	0	0	1
S_x	1010	111	0000	0	0	0	0
S_2	1111	110	1111	0	0	1	0

表 5.8 给出了 LUT（ROM）法和微程序法的比较。使用原 SM 图（图 5.13）的 ROM 法需要 2^9 个入口，因为其需要 3 个状态变量和 6 个输入。每个入口均为 7 位，3 位用于下一状态变量，4 位用于输出。双地址微代码的入口基于表 5.7，单地址微代码的入口基于表 5.6。

我们前面研究的实现 SM 图的方法是微程序法的例子。图 5.33 中的计数器类似于计算机中的程序计数器，它提供了下一条要执行的指令地址。ROM 输出一个微指令，由其他的硬件来执行。每个微指令都像一个条件转移指令，不断对输入进行检测，当检测到的输入为真时，则转移到不同的地址；否则按顺序执行下一条指令。微指令的输出域中含有控制硬件操作的位。

表 5.8　掷骰子游戏控制器的不同实现方法比较

方　法	ROM 大小	
	入口数×宽度	位　数
基于原 SM 图的 ROM 法	512×7	3584
双地址微代码	11×15	165
单地址微代码	12×11	132

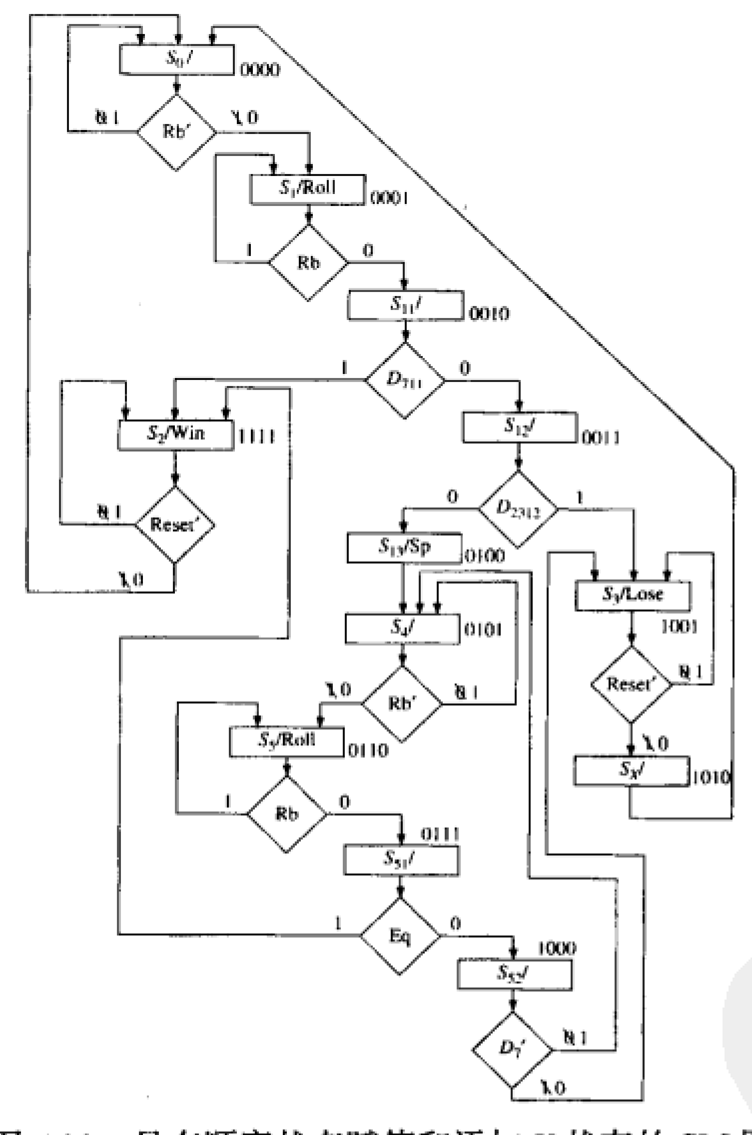


图 5.38　具有顺序状态赋值和添加 X 状态的 SM 图

5.6　链接状态机

当一个时序系统变得很大很复杂时，我们就希望可以将其拆分成为几个相互链接的小系统。而且每个小系统都很容易设计和实现。同样，某个子系统还可以在主系统的不同地方“调用”。这类似于把一个大的软件程序分成不同的子程序，而这些子程序可以随时被主程序调用。

图 5.40 给出了串行链接的两个状态机的 SM 图。主机（机器 A）按照顺序执行“一些状态”，

直到它要呼叫子机（机器 B）。当状态 S_A 到来时，输出信号 ZA 激活机器 B，机器 B 脱离空闲状态，并且执行一系列"其他状态"。当完成这些状态后，回到空闲状态之前，机器 B 发出 ZB 信号。当机器 A 接收到 ZB 信号后，它将继续执行"其他状态"。图 5.40 中假设这两个机器具有共同的时钟。

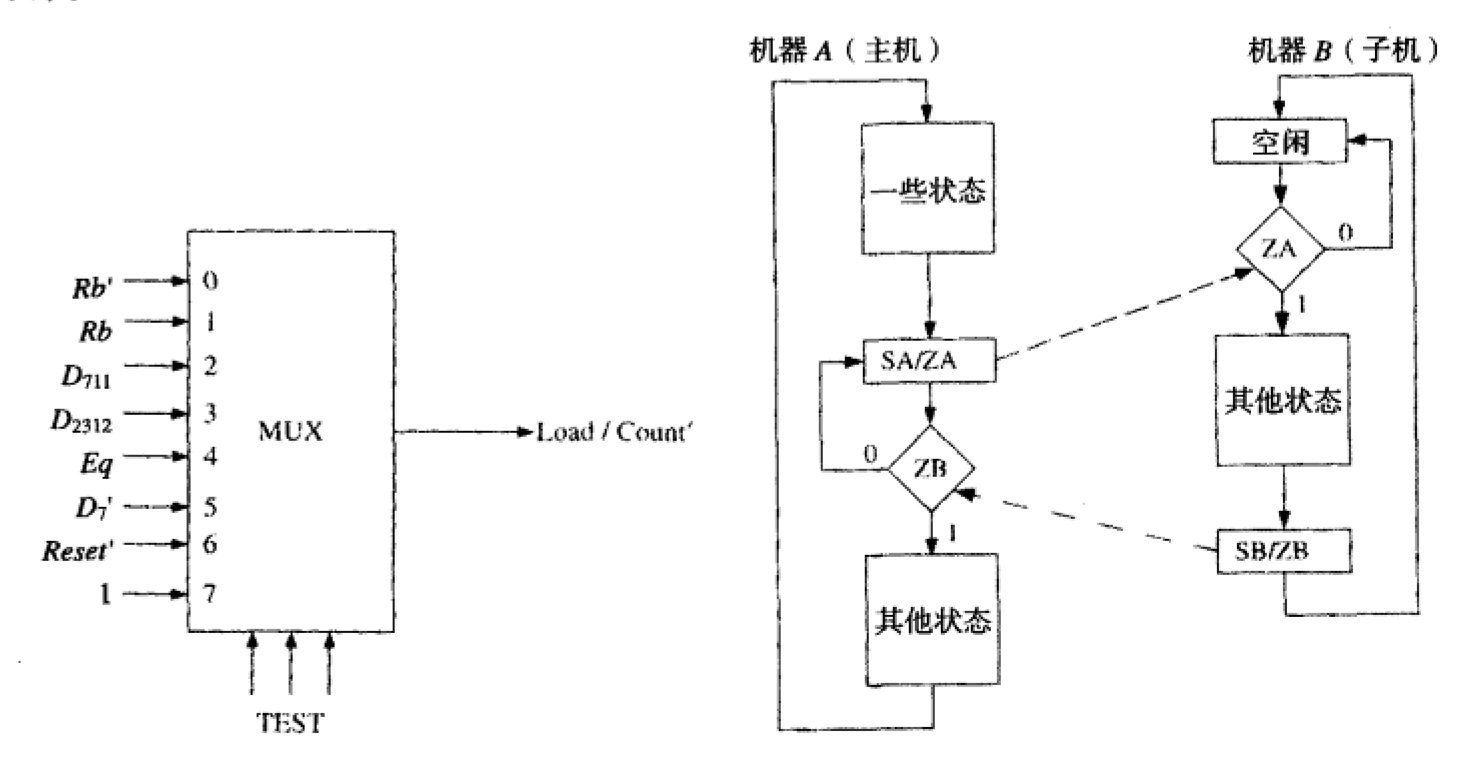
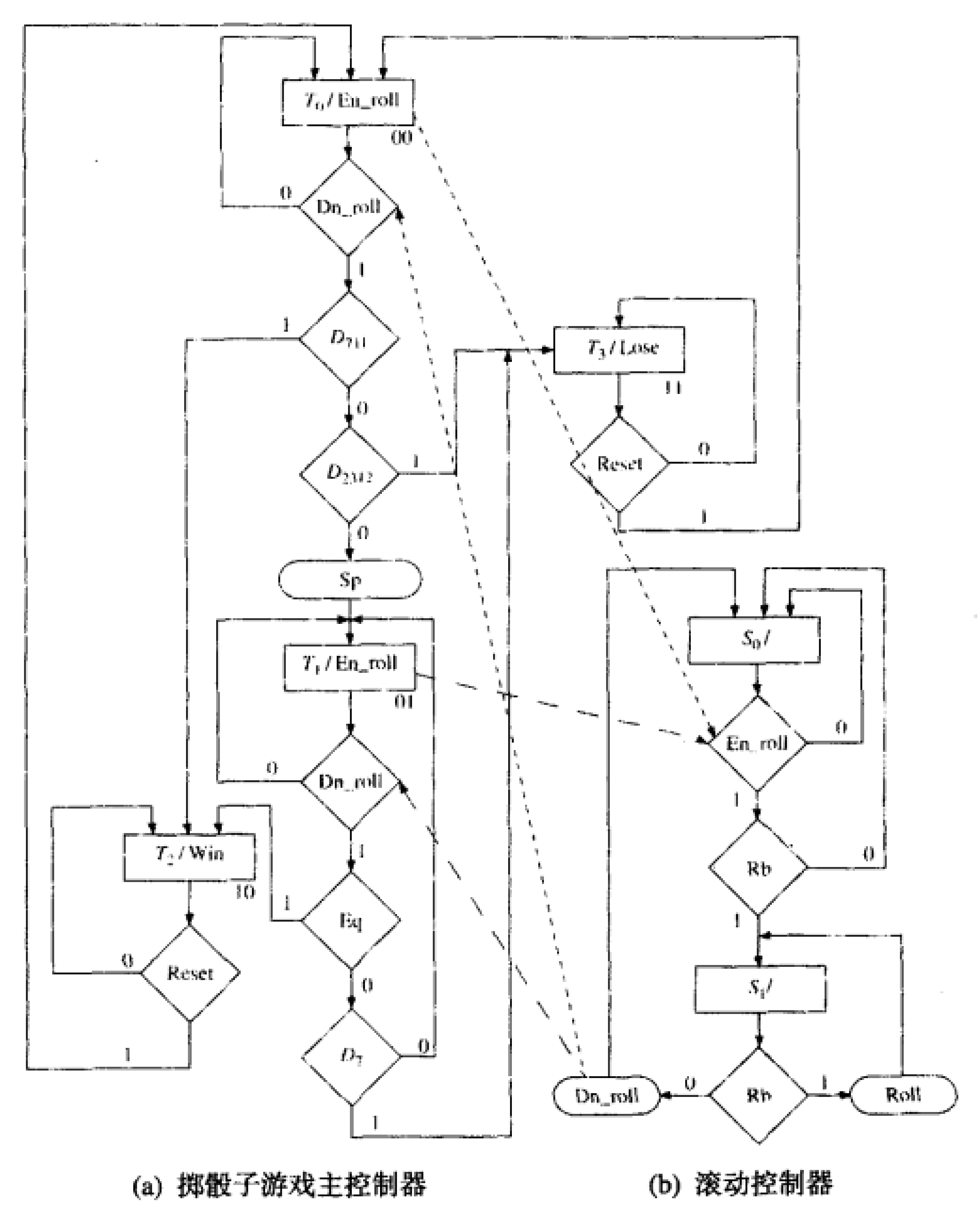


图 5.39 用单地址微代码实现骰子游戏时所用的 MUX　　图 5.40 串行链接状态机的 SM 图

作为使用链接状态机的一个例子，我们把图 5.13 中的 SM 图分为两个相连的 SM 图。在图 5.13 中，在状态 S_0 和 S_1 中，Rb 用来控制骰子的滚动，在状态 S_4 和 S_5 中也一样。由于该功能在两个地方重复出现，所以对滚动控制使用一个独立的状态机是合理的[图 5.41(b)]。使用单独的滚动控制可以使主骰子控制器[图 5.41(a)]的状态由 6 个减少为 4 个。主控制器在 T_0 生成信号 *En_roll*（滚动使能），并且在继续之前等待信号 *Dn_roll*（滚动完成）。主控制器在 T_1 也有相似的操作。滚动控制机在状态 S_0 等待，直到它从主投掷游戏控制器处获得信号 *En_roll*。然后，当滚动按钮被按下时（$Rb = 1$），滚动控制机到达状态 S_1 并且生成滚动信号（*Roll*）。滚动控制机将一直保持状态 S_1，直到 $Rb = 0$，这时生成 *Dn_roll* 信号，状态机回到状态 S_0。

在本章中，我们介绍了基于 SM 图的数字系统的设计过程。SM 图与状态图是等价的，但是通常通过分析 SM 图我们可以更容易地理解系统的运行过程。当我们画出一个数字系统的框图后，就可以用 SM 图描述其控制单元了。接下来我们可以根据该图用 VHDL 行为描述方式进行编程。通过使用 VHDL 语言编写的一个测试平台，我们可以对 VHDL 代码进行仿真来验证系统的功能是否符合要求。在对 VHDL 代码和 SM 图进行了必要的修正后，我们可以对系统进行详细的逻辑设计。重新编写 VHDL 程序的结构体并通过控制信号和逻辑表达式对系统的操作进行描述，我们可以对设计的正确性进行验证。

本章中我们还对实现控制单元的技术进行了讨论。我们主要介绍了两种技术：硬连线和微程序。我们介绍了如何根据 SM 图中的链路简单地得到逻辑表达式，还介绍了如何使用这些表达式简单地实现硬连线的控制单元。随后我们又介绍了微程序技术。在此技术中，控制字都被存储在微程序存储器中。微程序的大小可以通过变换 SM 图（使之每个状态只对一个限制量进行检测）来降低。对于复杂的系统，我们可以使用链接状态机将其控制器拆分为几个部分加以实现。



(a) 掷骰子游戏主控制器　　　(b) 滚动控制器

图 5.41　掷骰子游戏的链接 SM 图

习题

5.1 (a) 画出与下面状态表等价的 SM 图。在每个判断框只测量一个变量，并试着使判断框的数目最 少。
(b) 写出基于此 SM 图的状态机的 VHDL 描述。

当前状态	下一状态 $X_1X_2=$ 00	01	10	11	输出 (Z_1Z_2) $X_1X_2=$ 00	01	10	11
S_0	S_3	S_2	S_1	S_0	00	10	11	01
S_1	S_0	S_1	S_2	S_3	10	10	11	11
S_2	S_3	S_0	S_1	S_1	00	10	11	01
S_3	S_2	S_2	S_1	S_0	00	00	01	01

5.2 画出与下面状态表等价的 SM 图。在每个判断框中只检测一个变量，并试着使用最少数目的判断框。要求在 SM 图上显示 Mealy 和 Moore 输出。

当前状态	下一状态 $X_1X_2=$ 00	01	10	11	输出 $(Z_1Z_2Z_3)$ $X_1X_2=$ 00	01	10	11
S_0	S_1	S_1	S_1	S_1	000	100	110	010
S_1	S_1	S_1	S_0	S_0	001	001	001	001

5.3 一个委员会有 15 个具有投票资格的成员。每次召开董事会时参加成员的人数必须多于半数（8 个人，且 8 个人是可召开会议的最低限额人数），而且在讨论和投票决定某些事项时，必

须有 2/3 的成员参加。若满足最低限额人数，但是有偶数个成员（包括主席在内）投票时，主席可以投两票。会议室门上有三盏灯（绿、蓝、红）用来指示限额人数的状态。设计一个系统 SM 图，当满足限额人数时，绿灯亮；当讨论事项时，蓝灯亮；当限额人数满足但人数为偶数时，红灯亮。绿灯和红灯可以同时亮；绿灯、蓝灯和红灯可以同时亮。

假设只能通过一道门出入会议室。在此门内外两侧各有一个光电管（门内为 PHOTO1，门外为 PHOTO2）。当光束照到每个光电管时，输出为假；当光束被挡住时，输出为真。假设一旦有一个人开始进入，进程在另一个人进入或离开前完成（每次只有一个人进入或离开）。如果 PHOTO2 先为真，然后 PHOTO1 为真，则生成 *ENTER* 信号；若 PHOTO1 先为真，然后 PHOTO2 为真，则生成 LEAVE 信号。*LEAVE* 信号和 *ENTER* 信号不可同时为真。假设在对信号进行读取前，所有信号均为真。当对信号读取完毕后，再给门控制器提供一个信号（*READY*），指出另一个人可以从此门进入或离开。

(a) 画出此电路的数据部分的框图。假设 *ENTER* 和 *LEAVE* 信号可用（即此间中你不用生成它们）。

(b) 画出控制器的 SM 图。写出实现此设计的步骤，对所有使用的控制信号加以定义。

(c) 画出可以生成 *LEAVE* 和 *ENTER* 信号的电路的 SM 图。

5.4 (a) 一个除法器的被除数为 8 位，除数为 5 位，商为 3 位。当 $St = 1$ 时，被除数被载入。画出此除法器的框图。

(b) 画出控制电路的 SM 图。

(c) 基于此 SM 图写出 VHDL 程序。要求此程序中明确给出控制信号。

(d) 给出验证算术运算 93 除以 17 的仿真程序。

5.5 画出习题 4.13 中 BCD 码-二进制数转换器的 SM 图。

5.6 画出习题 4.14 中平方根电路的 SM 图。

5.7 画出习题 4.22 中二进制乘法器的 SM 图。

5.8 设计一个二进制码-BCD 码转换器，要求可以把一个 10 位二进制数转化为一个 3 数字 BCD 码，假设二进制数≤999，初始化时，二进制数放置在寄存器 *B* 中，当收到 *St* 信号时，开始转化为 BCD 码，转换后的 BCD 码存放在寄存器 *A* 中（12 位）。*A* 初始化为 0000 0000 0000。转换算法为：如果 *A* 中任意一个码字≥0101，则将此码字加上 0011，然后把寄存器 *A* 和 *B* 同时左移一位。重复以上过程 10 次。每一次其实都是对 BCD 码乘以 2（左移），再对其与下一个二进制数求和。

(a) 用此算法把 100011101 转换为 BCD 码。

(b) 画出此二进制-BCD 码转换器的实现框图，移位数由一个计数器记录，且移位 10 次后计数器输出信号 C_{10}。

(c) 画出控制器的 SM 图（3 个状态）。

(d) 写出控制器的 VHDL 程序。

5.9 设计一个 16 位二进制整数乘法器（此设计与图 4.33 和图 4.34 相似）。

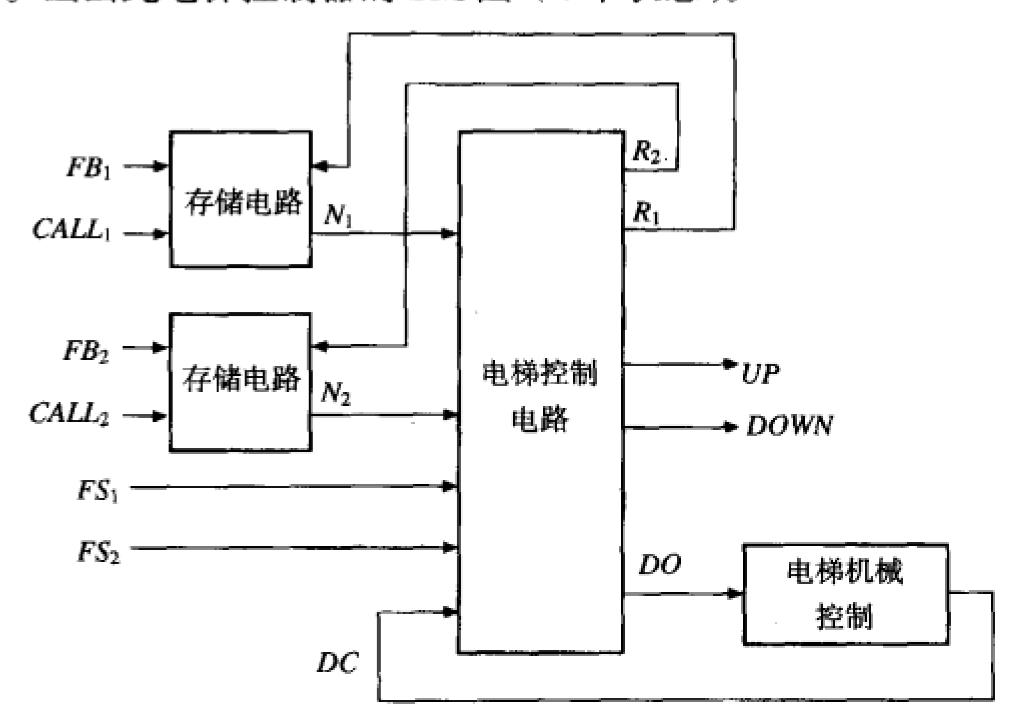
(a) 画出实现框图，在控制电路中加入一个计数器对移位数进行计数。

(b) 画出控制器的 SM 图（3 个状态），假设 15 次移位后计数器输出 $K = 1$。

(c) 写出此设计的 VHDL 程序。

5.10 一个两层楼的电梯的控制器的框图如下所示。输入 FB_1，FB_2 为电梯内的楼层按键。输入 $CALL_1$, $CALL_2$ 是大厅内的呼叫按键。输入 FS_1, FS_2 是楼层开关，当电梯停在一楼时 $FS = 1$，

停在二楼时 $FS_2 = 1$。输出 $UP, DOWN$ 控制发动机，当 $UP = DOWN = 0$ 时，电梯停。N_1, N_2 是触发器，指示何楼层需要电梯。R_1, R_2 是触发器的复位信号。$DO = 1$ 时电梯门开启，$DC = 1$ 时电梯门关闭。画出此电梯控制器的 SM 图（4 个状态）。



5.11 为习题 5.10 中电梯控制器设计一个测试平台。此程序要求有两个进程：一个用于模拟电梯的操作（包括电梯门的操作），另一个提供一串按键信号序列用以测试控制器的运行。

模拟电梯操作：若电梯在 1 层（$FS_1 = 1$）且接收到 UP 信号，则等待 1 s 后 FS_1 置 1；接着等待 10 s 后把 FS_2 置 1。此过程模拟电梯从 1 楼移到 2 楼。若电梯在 2 层（$FS_2 = 1$）且接收到 $DOWN$ 信号，则模拟过程相似。当接收到 $DO = 1$ 时，电梯门打开；当电梯门关闭（$DC = 0$）5 s 后，再把 DC 置 1。

测试序列：$CALL_1$, 2, FB_2 , 4, FB_1 , 1, $CALL_2$, 10, FB_2 。

假设每次按键均持续 1 s 后再放开，按键之间的数字表示两次按键之间的延迟（按秒计算）；此延迟要加到 1 s 的按键时间上。

完成下面测试平台的程序。

```
entity test_el is
end test_el;

architecture eltest of test_el is
  component elev_control
    port(CALL1, CALL2, FB1, FB2, FS1, FS2, DC, CLK: in bit;
         UP, DOWN, DO: out bit);
  end component;
```

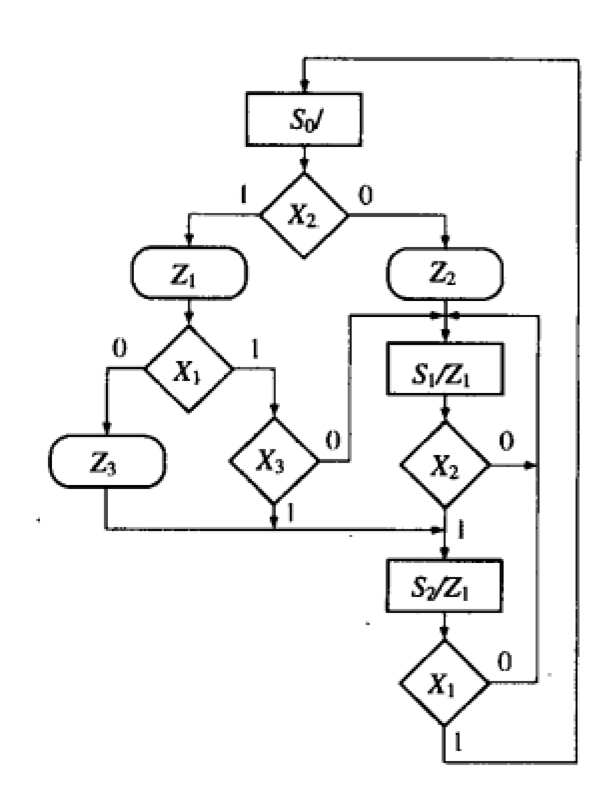
5.12 针对下面的 SM 图：

(a) 画出时序图，给出时钟、状态(S_0, S_1, S_2)、输入(X_1, X_2, X_3)和输出。输入序列为 $X_1X_2X_3 = 011$, 101, 111, 010, 110, 101, 001。假设输入在时钟脉冲中间改变，状态在时钟上升沿发生改变。

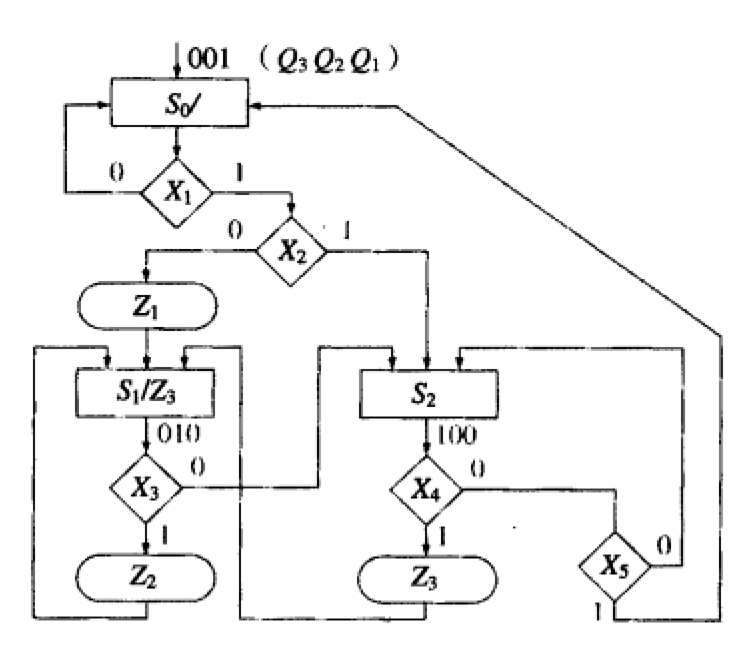
(b) 状态赋值为 S_0: $AB = 00$; S_1: $AB = 01$; S_2: $AB = 10$。求下一状态和输出的逻辑表达式，并用随意项($AB = 11$)进行状态化简。

(c) 用 PLA 和一个 D 触发器实现此 SM 图，并给出 PLA 表。

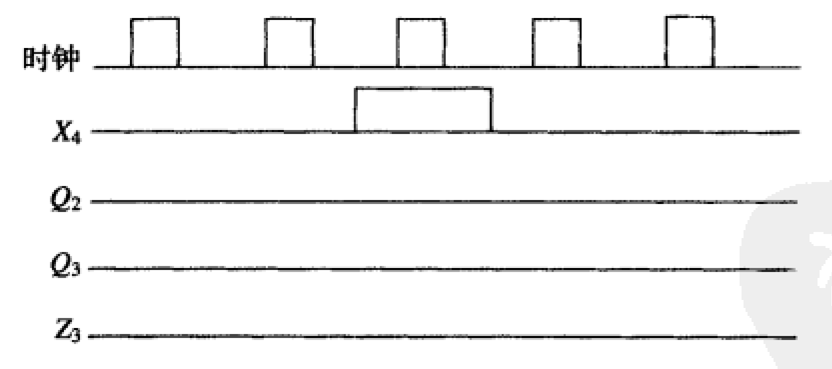
(d) 如果用 ROM 代替 PLA，那么要使用多大的 ROM？给出 ROM 表的前 5 行。



5.13 针对下面的 SM 图：



(a) 完成下面的时序图（假设 $X_1 = 1, X_2 = 0, X_3 = 0, X_5 = 1$，$X_4$ 如图中所示）。触发器在时钟下降沿发生状态改变。



(b) 使用已知的 one-hot 状态赋值，并由 SM 图得到最小化的下一状态和输出的逻辑表达式。

(c) 写出此数字系统的 VHDL 程序。

5.14 (a) 画出与图 4.46 状态图等价的 SM 图。

(b) 如果用一个 PLA 和 3 个触发器(A, B, C)实现此 SM 图，请给出 PLA 表（状态转移表），要求使用直接二进制状态赋值。

(c) 给出由 PLA 表确定的 A^+ 的表达式。

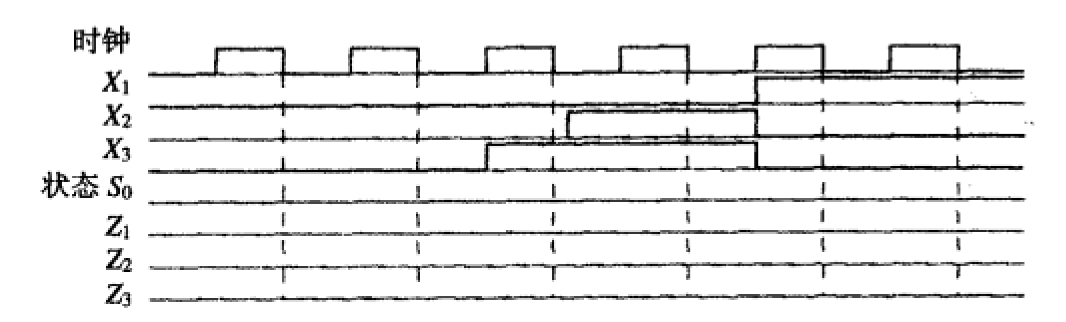
(d) 若使用 one-hot 状态赋值，给出输出和下一状态表达式。

5.15 (a)写出描述下面 SM 图的 VHDL 程序，假设在时钟下降沿发生状态改变，程序中要求使用两个进程。

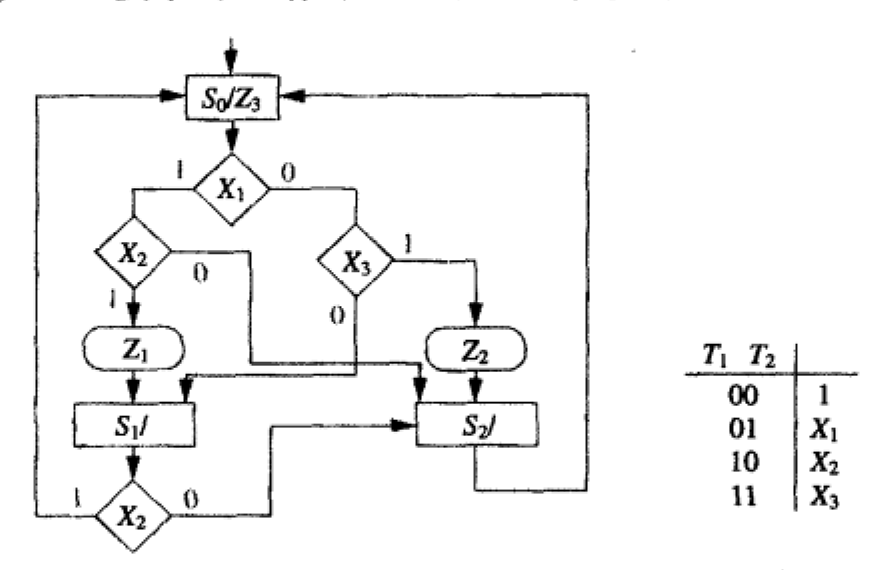
(b) 用一个 PLA 和两个触发器（A 和 B）实现此 SM 图，完成下面的状态转移表（PLA 表），并根据此 PLA 表给出 A^+ 的表达式。

A	B	X_1	X_2	X_3	A^+	B^+	Z_1	Z_2	Z_3

(c) 完成下面的时序图。



5.16 用 PLA（要求用最少的输入）、一个多路选择器、一个可载入数据的计数器（如 74163）实现下面的 SM 图。PLA 要求可以给出 NST 和 NSF，多路选择器的输入见下表。



T_1 T_2	
00	1
01	X_1
10	X_2
11	X_3

(a) 画出框图。

(b) 把此 SM 图转换为正确的形式，要求尽量少的加入其他的状态。

(c) 进行状态赋值并给出 PLA 表的前 5 行。

(d) 写出此系统使用 PLA 实现时的 VHDL 程序。

5.17 用图 5.29 中的双地址微程序结构实现习题 5.16 的 SM 图。

(a) 把此 SM 图转换为正确的形式，要求尽量少的加入其他的状态，并进行状态赋值。

(b) 写出实现此电路所需的微程序。

(c) 给出微程序法所需的 ROM 大小。

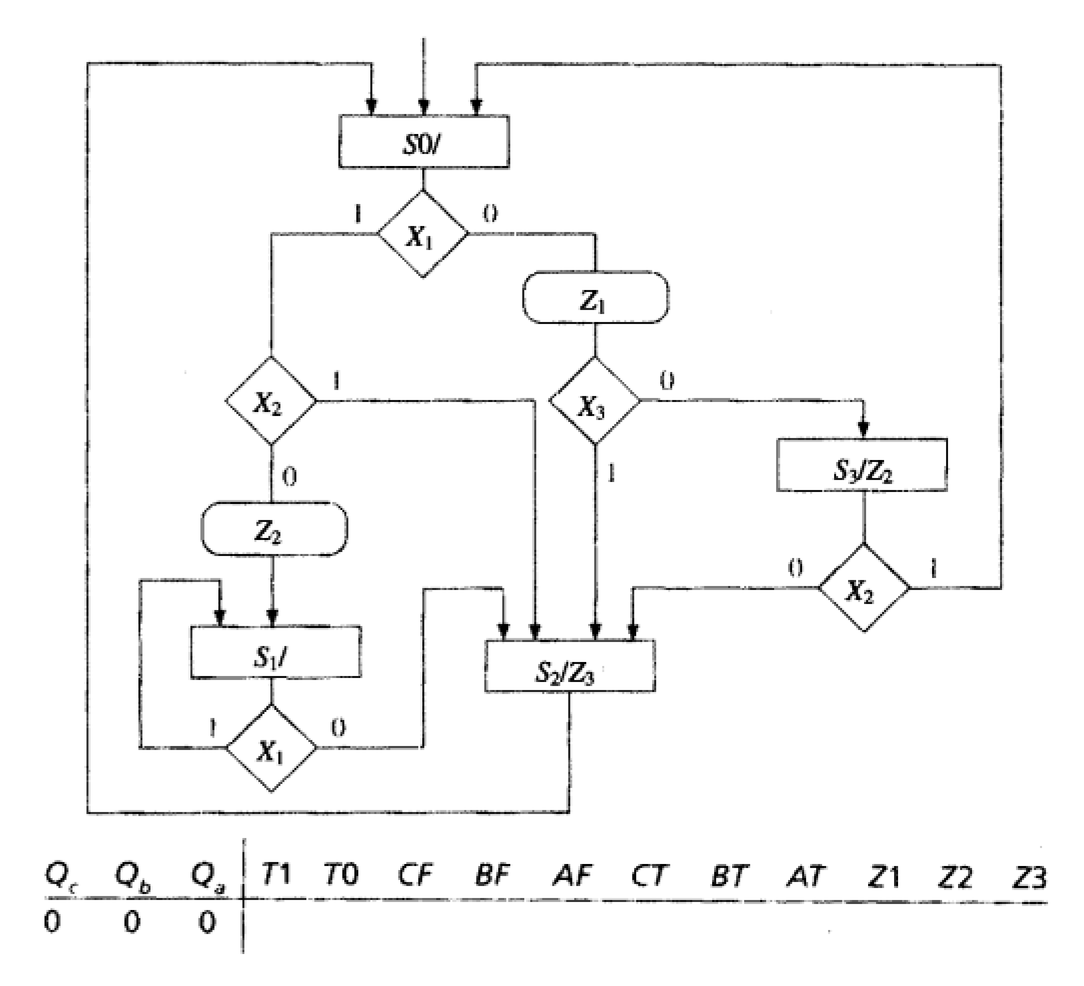
(d) 如果不使用微程序法，而使用传统 ROM 方法实现此 SM 图，则所需 ROM 的大小为多少?

5.18 下面的 SM 图将用如图 5.29 所示的双地址微程序结构实现。

(a) 通过添加最少的状态，把此 SM 转换为另一种形式，并选用一种适合的状态赋值。

(b) 写出实现此 SM 图的微代码。

(c) 画出用 ROM, MUX 和触发器实现此 SM 图的框图。



Q_c	Q_b	Q_a	T1	T0	CF	BF	AF	CT	BT	AT	Z1	Z2	Z3
0	0	0											

5.19 (a) 若用带有计数器、ROM 和多路选择器的单地址微程序结构（参见图 5.33）实现 SM 图，则需要满足 SM 图中的哪些条件?

(b) 如果习题 5.16 中的 SM 图用此种结构的微程序系统实现，请给出修改后的 SM 图和所需的状态赋值。

5.20 (a) 若用带有计数器、ROM 和多路选择器的双地址微程序结构（见图 5.33）实现 SM 图，则需要满足 SM 图中的哪些条件?

(b) 如果习题 5.18 中的 SM 图用此种结构的微程序系统实现，请给出修改后的 SM 图和所需的状态赋值。

5.21 用一个 PLA、一个 4 选 1 MUX 和一个计数器实现下面的 SM 图。

(a) 画出框图，并给出 MUX 的输入。

(b) 把此 SM 图转化为一个适当的形式，并在下图中标示出所做的必要的改动。

(c) 进行恰当的状态赋值，并给出 POM 表的前 6 行。

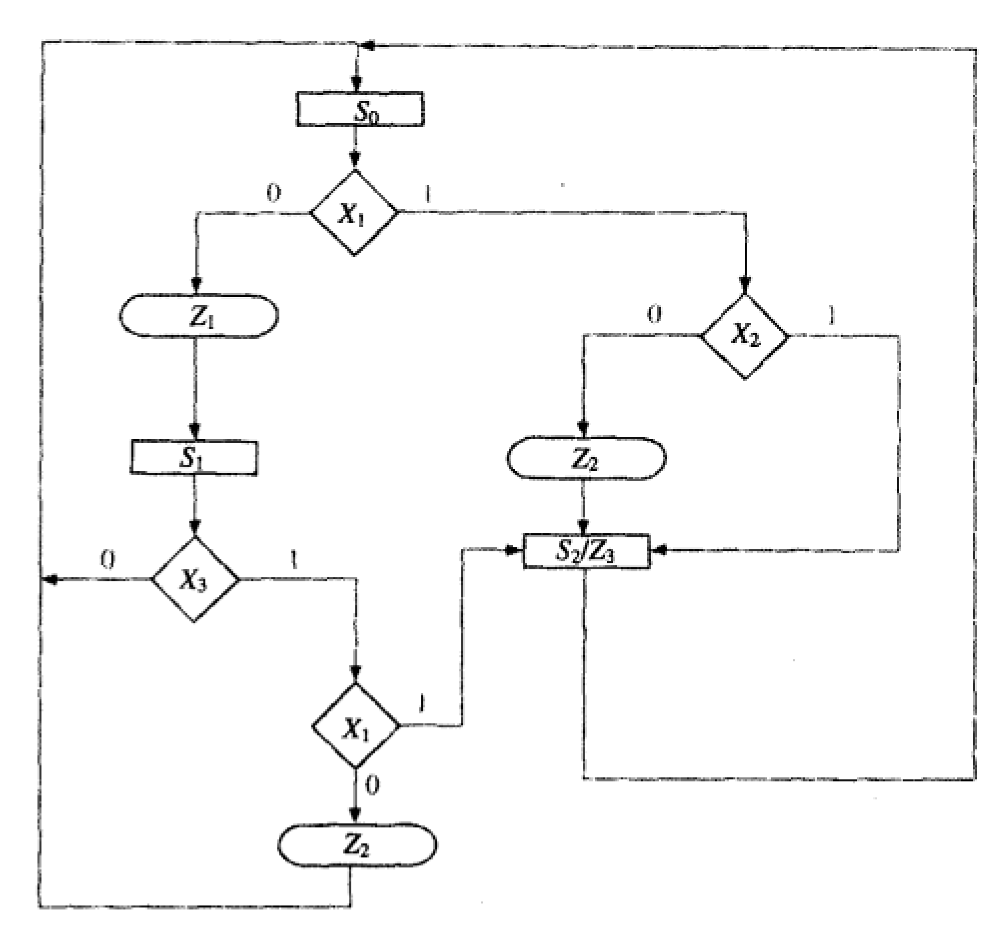
5.22 用图 5.29 中的双地址微程序结构实现习题 5.21 的 SM 图。

(a) 把此 SM 图转化为一个恰当的形式，要求尽量少的加入其他的状态。需要做何改变?

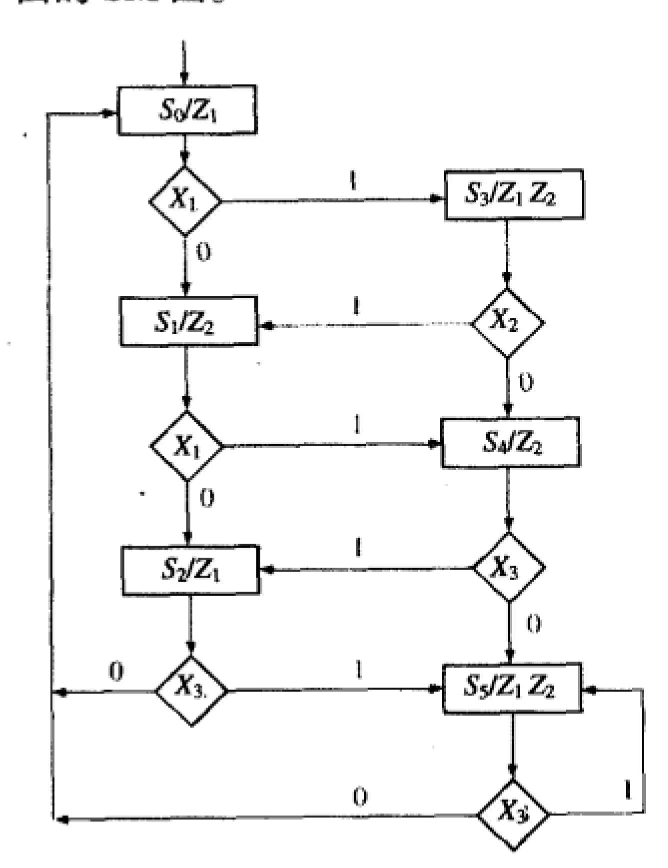
(b) 写出用所指定的硬件实现此状态机的微代码。在微代码中，你可以使用状态名 S_0, S_1, S_2 等，而不用使用 1 位状态赋值（0 或 1）。

(c) 实现此微代码需要使用多大的 ROM? 解释一下你是如何计算的。

(d) 若用原始 ROM 法实现所给出的 SM 图（未改变前的），则需要使用多大的 ROM? 解释一下你是如何计算的。



5.23 用单地址微程序实现下面的 SM 图。



(a) 给出新的 SM 图和新的状态赋值。MUX 输入为 $1, X_1, X_2$ 和 X_3。输入不可以取反，如果需要可以加入状态。

(b) 写出用单地址微程序实现此状态机的微代码。

(c) 若用原始 ROM 法实现给出的 SM 图（未改变前的），则需要使用多大的 ROM？解释一下你是如何计算的。

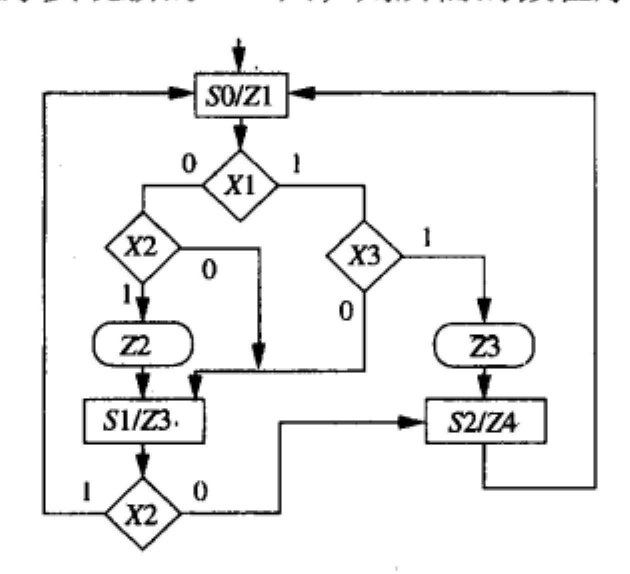
5.24 已知 SM 图如下所示。

(a) 给出下一状态和输出表达式，假设状态赋值为 $S_0 = 00, S_1 = 01, S_2 = 10$。

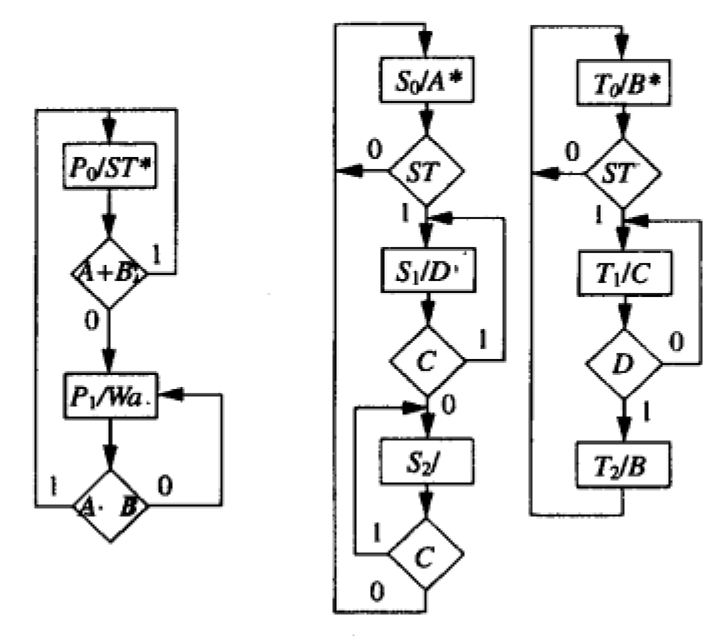
(b) 若用单地址微程序实现此 SM 图，且此微程序中只给出 NST，则下面的 SM 图应做何修改？给出新的 SM 图和新的状态赋值。

(c) 写出实现此电路所需的单地址微程序。

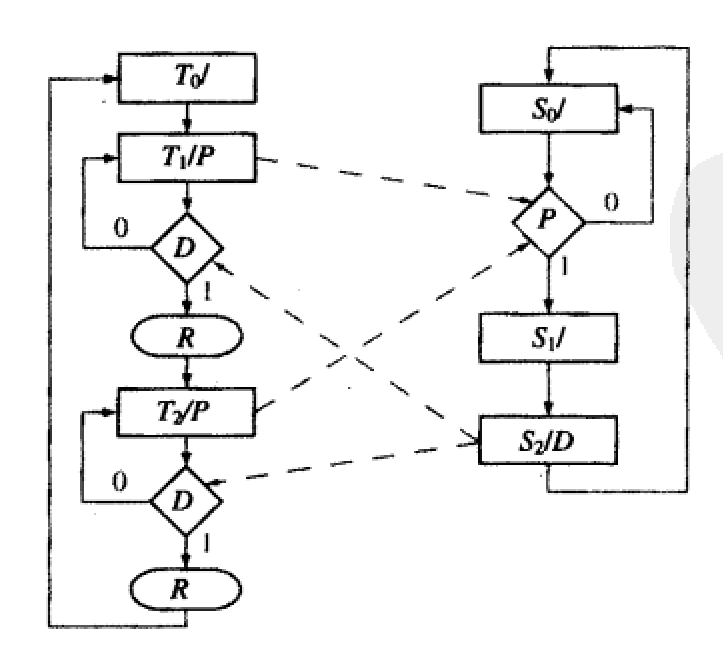
(d) 如果要用单地址微程序实现新的 SM 图，则所需的微程序的 ROM 的大小为多少？



5.25 现有三个相互连接的状态机，其 SM 图如下所示。假设所有状态在时钟下降沿到来时发生改变。给出时序图（包括 *ST, Wa, A, B, C, D*）。状态机均从带有星号（*）的状态开始执行。



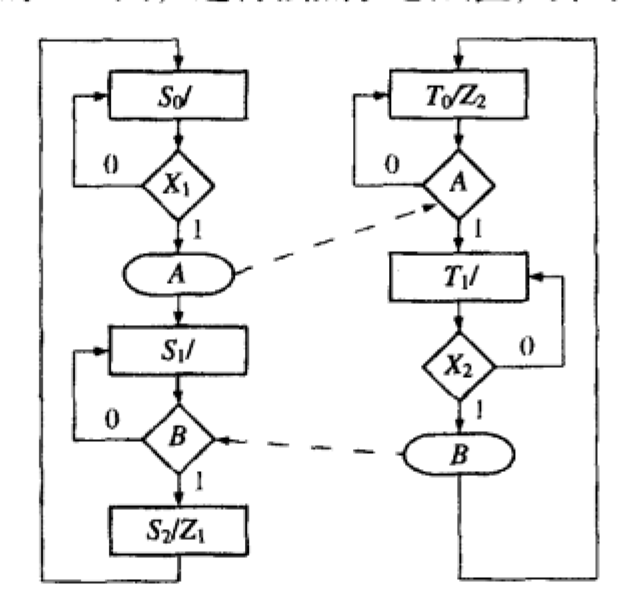
5.26 两个链接状态机的 SM 图如下所示。机器 T 的起始状态为 T_0，机器 S 的起始状态为 S_0。画出时序图，要求显示 CLK, T 和 S 的状态、10 个时钟周期内信号 P, R 和 D。状态改变均发生在时钟上升沿。

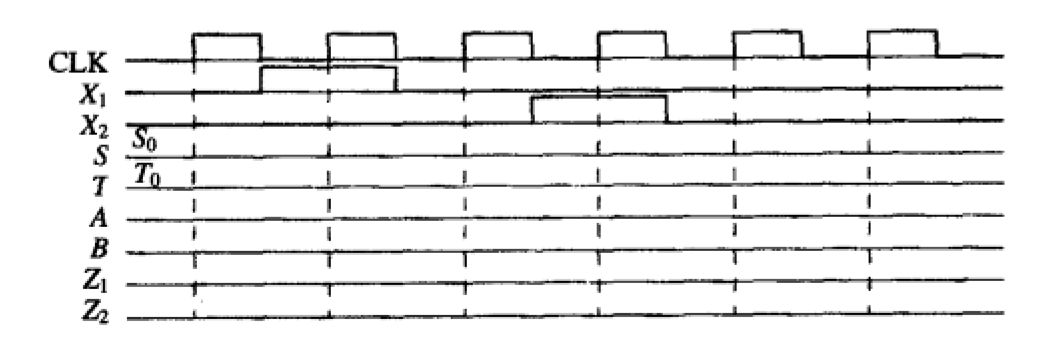


5.27 两个链接状态机的 SM 图如下所示。

(a) 完成下面的时序图。

(b) 对于左边的 SM 图，进行独热状态赋值，并导出 D 触发器输入表达式和输出表达式。





第 6 章　FPGA 设计实例

本章对使用 FPGA 实现数字系统的相关内容加以介绍。我们列举一些简单的设计例子，把它们手工映射到 FPGA 的基本构建模块中去，说明这些基本模块的结构在实现数字系统时的权衡问题。当我们需要把大型逻辑表达式化为几个小型逻辑表达式时，就需要使用香农分解，本章对此也将进行介绍。我们讨论了 one-hot 状态赋值方法，它非常适合于像 FPGA 这类器件。我们介绍了设计流程，还对综合、映射和布局等方面也进行了简单的讨论。在讨论和例子中，我们涉及了几个商用 FPGA 的一些特性，但是我们并没有全面地介绍任何商用的某一 FPGA 家族的体系结构，只介绍其一般意义上的基本原理。一旦掌握了基本原理之后，你就可以通过生产商提供的技术手册或网上的相关信息，能够对将要使用的特定器件进行详细了解。

6.1　FPGA 中的函数实现

一般来说，基于 VHDL 或 Verilog 等硬件描述语言生成行为描述、RTL 或结构描述的设计模块，用 CAD 软件来自动实现该设计在 FPGA 中的综合、映射、分区、布局和布线。为了了解如何把一个设计分配到 FPGA 中去，我们用 FPGA 设计几个简单系统。

假设我们要设计一个 4 选 1 多路选择器，要用的 FPGA 的逻辑模块如图 6.1(a)所示。这一模块由两个 4 变量函数生成器 X 与 Y 和两个触发器构成。X 函数生成器可以生成关于 X_1, X_2, X_3 和 X_4 的任意函数。同理，Y 函数生成器可以生成关于 Y_1, Y_2, Y_3 和 Y_4 的任意函数。函数的输出可以直接或锁存后送到输出逻辑模块。锁存输出为 QX 和 QY，组合输出为 X 和 Y。假设多路选择器的输入为 I_0, I_1, I_2 和 I_3，选择控制为 S_1 和 S_0，则多路选择器的输出函数可以写为

$$M = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3 \tag{6.1}$$

一个 4 选 1 多路选择器可以分解为三个 2 选 1 多路选择器，如图 6.1(b)所示。

$$M_1 = S_0'I_0 + S_0I_1$$

$$M_2 = S_0'I_2 + S_0I_3$$

第三个 2 选 1 多路选择器必须用来生成 4 选 1 多路选择器的输出：

$$M = S_1'M_1 + S_1M_2$$

这一输出与原 4 选 1 多路选择器(M)的输出是相同的。前两个 2 选 1 多路选择器（M_1 和 M_2）可以在一个逻辑模块中实现，第三个 2 选 1 多路选择器(M)可以在第二个逻辑模块中实现。这样，实现一个 4 选 1 多路选择器就需要两个这种类型的逻辑模块。第一个逻辑模块生成函数为

$$X = M_1 = S_0'I_0 + S_0I_1$$

$$Y = M_2 = S_0'I_2 + S_0I_3$$

第二个逻辑模块只使用了一半，X 函数生成器产生函数为

$$M = S_1'M_1 + S_1M_2$$

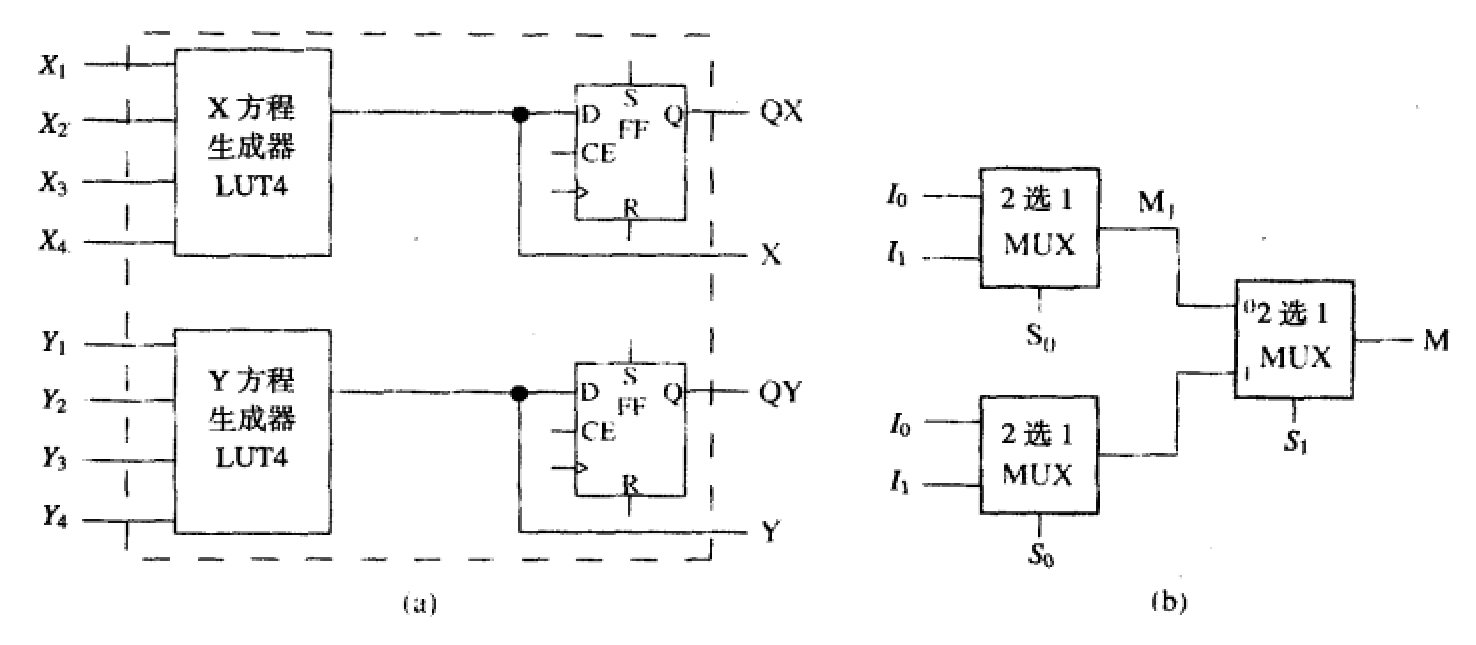


图 6.1 (a)FPGA 构建模块示例 3；(b)用两个 2 选 1 MUX 构成一个 4 选 1 MUX

M_1 和 M_2 使用的路径在图 6.2 中用高亮线表示。该设计中没有使用触发器。

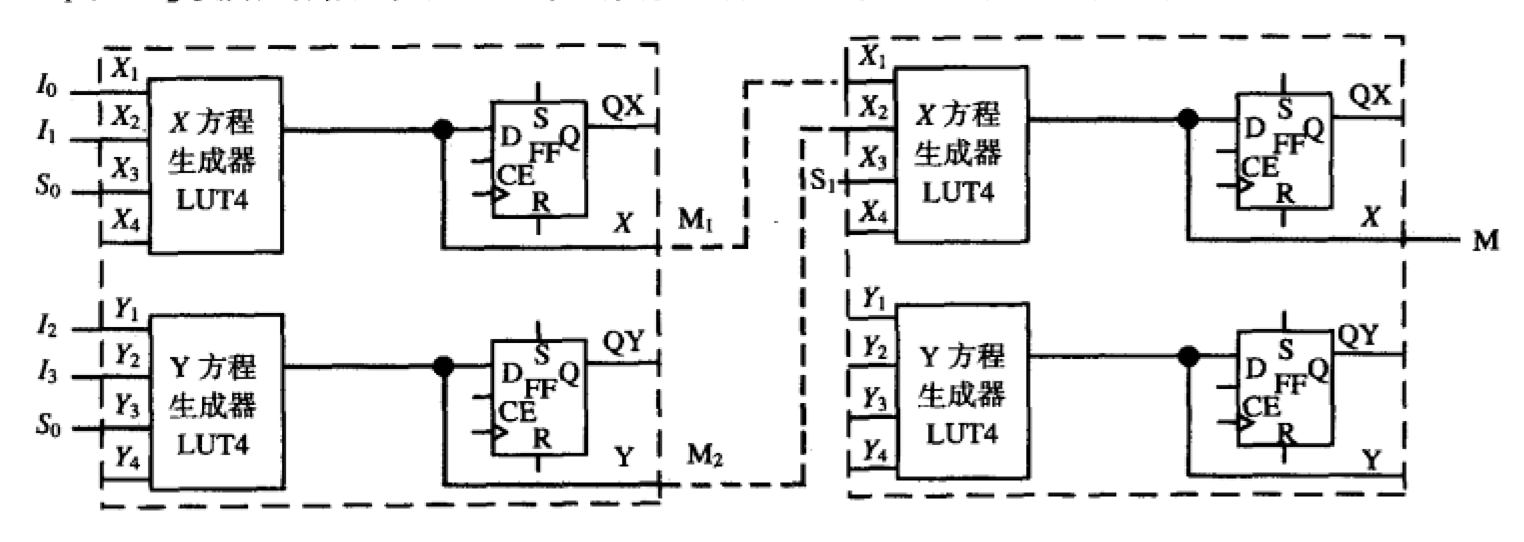


图 6.2 形成 4 选 1 多路选择器的高亮路径

许多现代 FPGA 都使用 4 输入查找表(LUT)作为基本构建模块。很多设计者把该模块称为 LUT4，它可以用来实现任意 4 变量函数。为了用 SRAM 技术实现 4 输入 LUT，要使用 SRAM 的 16 位。

例 用 LUT4 实现图 6.2 中的多路选择器时，LUT 表中的内容是什么？

解：如图 6.2 所示，要使用三个查找表实现 M_1，M_2 和 M 函数，它们都是 2 选 1 多路选择器。

输入				输出
X_4	$X_3(S_0)$	$X_2(I_1)$	$X_1(I_0)$	X
x	0	0	0	0
x	0	0	1	1
x	0	1	0	0
x	0	1	1	1
x	1	0	0	0
x	1	0	1	0
x	1	1	0	1
x	1	1	1	1

假设 X_1 和 Y_1 是 LUT 地址的最低有效位，X_4 和 Y_4 是 LUT 地址的最高有效位，则可以写出每个 LUT 的真值表，如上图所示。当 S_0 为 0 时，输出(X)等于 I_0；当 S_0 为 1 时，输出等于 I_1。设三个 LUT 分别为 LUT-M1, LUT-M2 和 LUT-M。

每个 LUT 的最高有效位均未使用。LUT 中的前 8 个数据可以在后 8 个中重复使用，这是因为不管 X_4 取什么值，我们希望完成 2 选 1 多路选择器的功能。因此，LUT-M1 中的内容为

LUT-M1–0，1，0，1，0，0，1，1，0，1，0，1，0，0，1，1

由于图 6.2 中三个 LUT 均用来实现 2 选 1 多路选择器，所以按图中的输入连接它们的内容是完全相同的。第二个和第三个 LUT 的内容为

LUT-M2–0，1，0，1，0，0，1，1，0，1，0，1，0，0，1，1

LUT-M–0，1，0，1，0，0，1，1，0，1，0，1，0，0，1，1

有些 FPGA 的基本模块提供两个 4 变量函数生成器和两个函数输出的组合方法。参阅图 6.3 中的逻辑模块。该可编程逻辑模块有 9 个逻辑输入（$X_1, X_2, X_3, X_4, Y_1, Y_2, Y_3, Y_4$ 和 C），它可以生成两个独立的 4 变量函数：

$$f_1(X_1, X_2, X_3, X_4) \text{ 和 } f_2(Y_1, Y_2, Y_3, Y_4)$$

该逻辑模块还可以生成一个基于 f_1, f_2 和 C 的函数 Z。模块中使用多个可编程多路选择器以决定输出为组合输出（Xout, Yout）还是时序输出（QX, QY）。该模块还可以生成任意 5 变量函数，函数形式为 $Z = f_1(F_1, F_2, F_3, F_4)\cdot C' + f_2(F_1, F_2, F_3, F_4)\cdot C$。当然，它也可以生成一些 6, 7, 8 甚至是 9 变量的函数。Xilinx 过去生产的 FPGA（XC4000）就具有类似的逻辑模块结构。

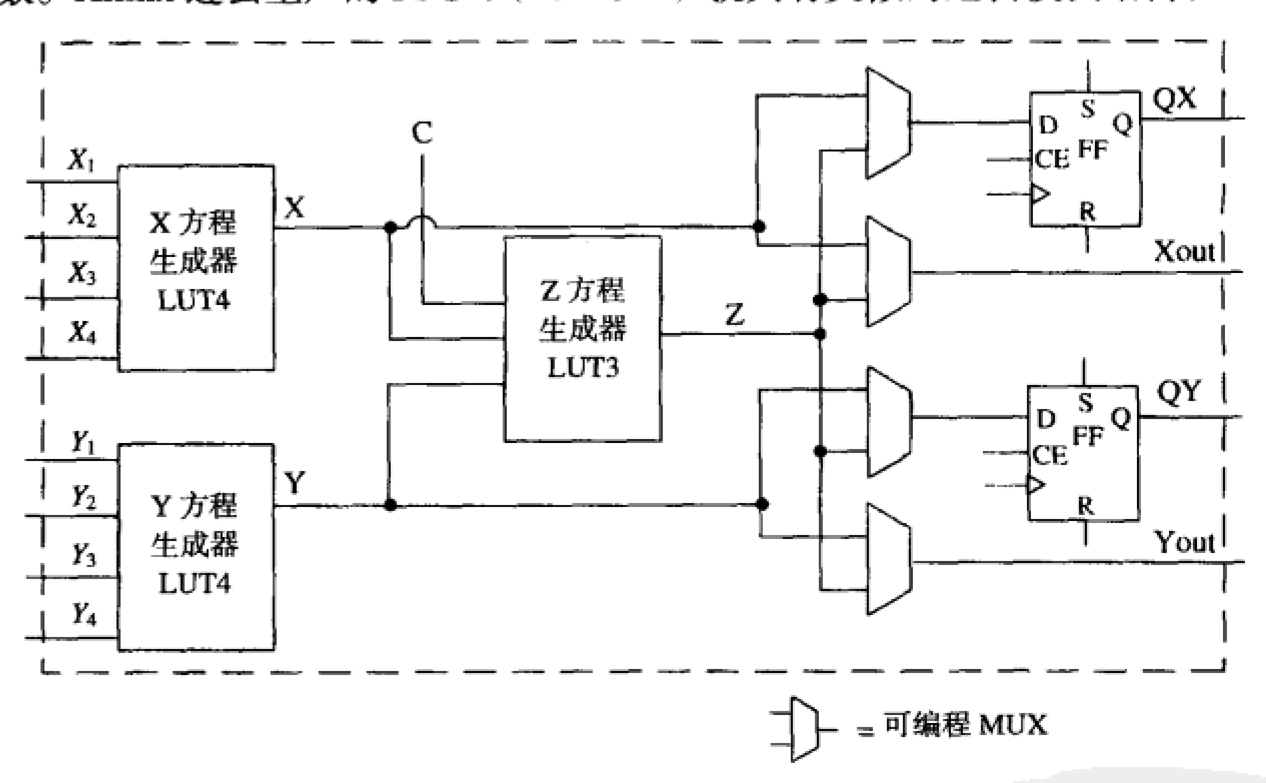


图 6.3　有三个 LUT 表的可编程逻辑模块示例

下面考虑如何用这一 FPGA 构建模块实现 4 选 1 多路选择器。我们用这种 FPGA 的一个逻辑模块就可以实现 4 选 1 多路选择器，如图 6.4 中高亮线所示。X 函数生成器实现函数 $M_1 = S_0'I_0 + S_0I_1$，Y 函数生成器（LUT4）实现函数 $M_2 = S_0'I_2 + S_0I_3$，Z 函数生成器实现函数 $M = S_1'M_1 + S_1M_2$。输入 C 连接选择信号 S_1 用于 Z 函数生成器中。该设计不需要使用触发器或锁存器。

通常，对于同一个设计有多种不同的映射方法。图 6.4 中的 4 选 1 多路选择器的实现用到了逻辑模块的输入信号 C。然而，该多路选择器的实现可以不使用输入信号 C。该多路选择器表达式[式(6.1)]的前两项中有 4 个变量：S_0, S_1, I_0 和 I_1；第三项和第四项中有 4 个变量：S_0, S_1, I_2 和 I_3。因此，我们可以用第一个 4 变量函数生成器实现头两项，用另一个 4 变量函数 生成器实现第三

项和第四项。但是，现在两个 4 变量函数生成器的输出需要组合，而 Z 函数生成器就可以用于这一目的。这种情况下，X 函数生成器（LUT4）产生函数：

$$F_1 = S_1' S_0' I_0 + S_1' S_0 I_1 \tag{6.1a}$$

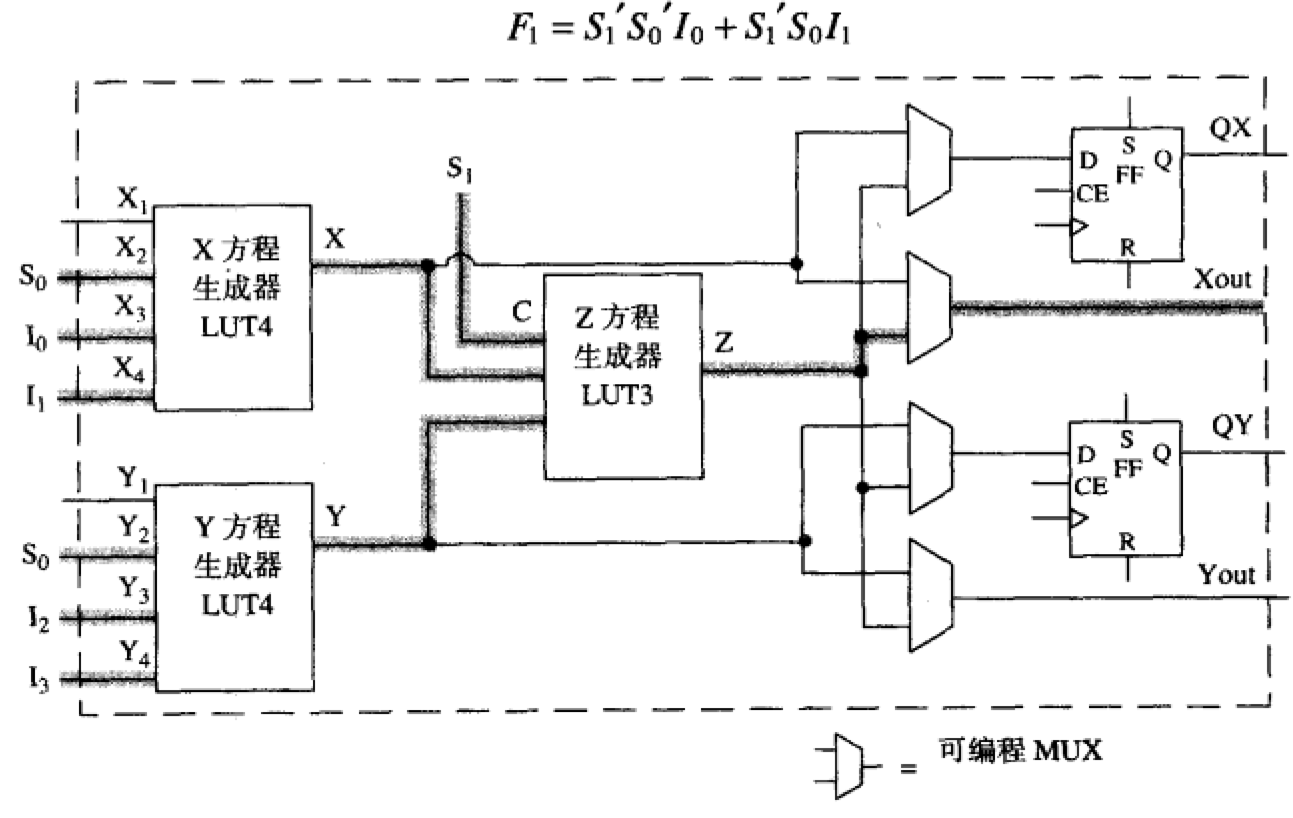


图 6.4　带有三个 LUT 表的可编程逻辑模块形成 4 选 1 MUX

这是函数表达式(6.1)的前一半。Y 函数生成器（LUT4）产生函数：

$$F_2 = S_1 S_0' I_2 + S_1 S_0 I_3 \tag{6.1b}$$

这是函数表达式(6.1)的后一半。Z 函数生成器（LUT3）完成函数 F_1 和 F_2 的或运算：

$$Z = F_1 + F_2 \tag{6.2}$$

此时，输入 C 就不需要了。此例说明了映射软件是如何根据目标技术的可用资源来进行电路映射的。

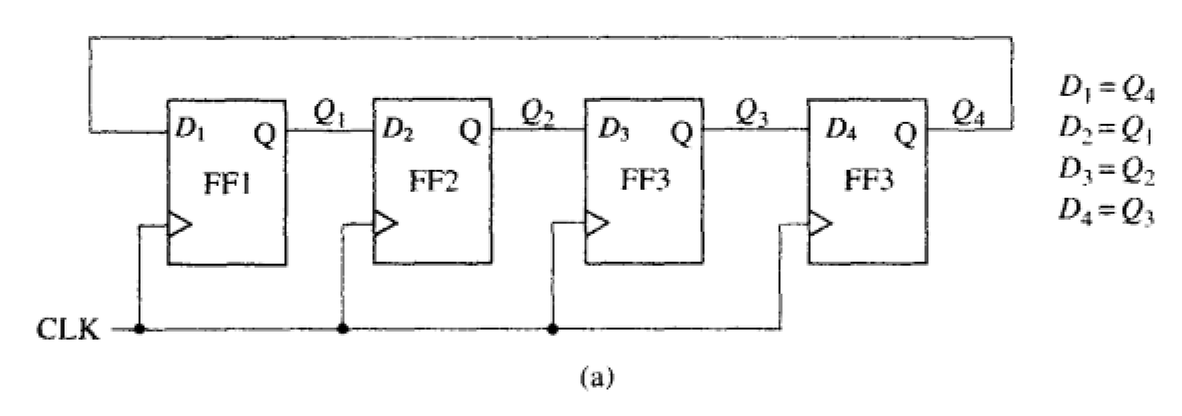
上面的例子说明使用 LUT 实现多路选择器代价很高，要使用三个函数生成器（LUT）才可以实现一个 4 选 1 多路选择器。由于每个 4 变量函数生成器均需要 16 个 SRAM 储存单元，所以如果用图 6.2 所示 FPGA 的构建模块来实现一个 4 选 1 多路选择器，就要使用 48 个储存单元。

实现一个 3 变量函数生成器（LUT3）需要 8 个储存单元。因此，图 6.3 所示多路选择器共需要 40 个储存单元（X 和 Y 各需要 16 个，Z 需要 8 个）。这些储存单元的内容就是我们编程时要下载到 FPGA 中去的部分数据。

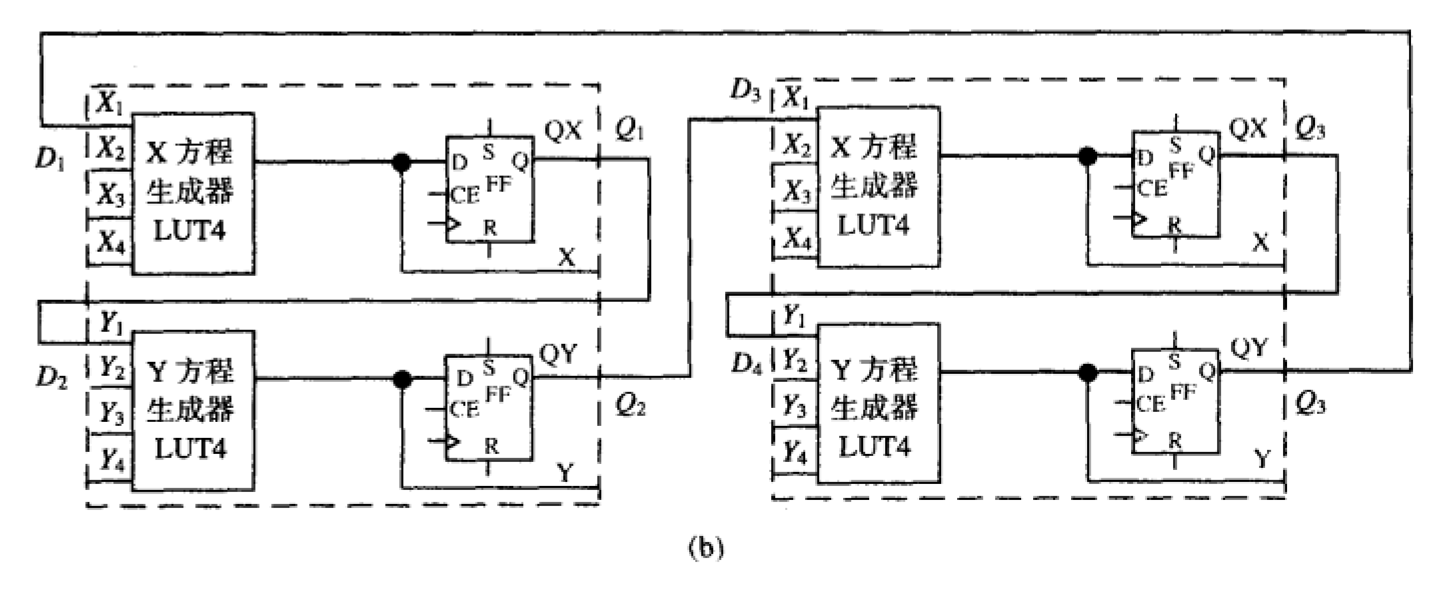
当 FPGA 中的可编程逻辑模块是可以实现复杂多变量函数的大模块时，有可能每个逻辑模块的很大一部分就用不上。我们举一个例子。假设我们必须要在 FPGA 中设计一个 4 位循环移位寄存器，该 FPGA 的构建模块类似于图 6.1(a)。在循环移位寄存器中，其最右边触发器的输出要反馈回最左边触发器的输入端，我们把这种移位寄存器称为环形计数器。由于一个 4 位移位寄存器需要 4 个触发器，所以实现该电路需要两个基本构建模块。4 个下一状态函数为 $D_1 = Q_4, D_2 = Q_1, D_3 = Q_2, D_4 = Q_3$。两个下一状态函数可以用一个模块中的组合函数生成器实现。图 6.5(b)中的高亮线为该循环移位寄存器的有效路径。X 函数生成器用来实现 $D_1 = Q_4$；Y 函数生成器用来实现 $D_2 = Q_1$。

注意，由于本例中触发器的下一状态函数很简单（它们只与前一个触发器的当前状态有关），所以 4 变量函数生成器有很大一部分没有使用。然而，一个函数生成器即使用于实现一个变量函

数，但该函数生成器的剩余部分也不能用于实现其他函数了。



(a)



(b)

图 6.5 (a)移位寄存器电路；(b)使用简单的 FPGA 构建模块实现(a)

例 如果要实现一个 3-8 译码器，那么需要多少个可编程逻辑模块[如图 6.1(a)所示]?

解：3-8 译码器有 3 个输入和 8 个输出。每个输出均需要一个 3 变量函数生成器。因为图 6.1(a)中的逻辑模块只能提供 4 变量函数生成器，因此只能用这一函数生成器实现一个输出函数。所以，实现一个 3-8 译码器一共需要 8 个函数生成器（比如，8 个 4 输入 LUT）。一个图 6.1(a)所示的逻辑模块可以实现两个输出，因此实现一个 3-8 译码器一共需要 4 个这种可编程逻辑模块。

如果 LUT 是基于 SRAM 的，那么在使用基于 LUT 的 FPGA 实现一个 3-8 译码器时，一共需要 128 个 SRAM 储存单元。如果用逻辑门实现该电路，那么只需要使用 3 个 3 输入与门和 3 个反相器。因此，在实现这种函数时，使用 LUT 代价很高。

有些 FPGA 使用多路选择器和门电路作为基本构建模块。有些 FPGA（如 Xilinx Spartan）可以提供 LUT 和多路选择器。映射软件根据目标技术的可用资源把设计映射到构建模块中。

6.2 基于香农分解的函数实现

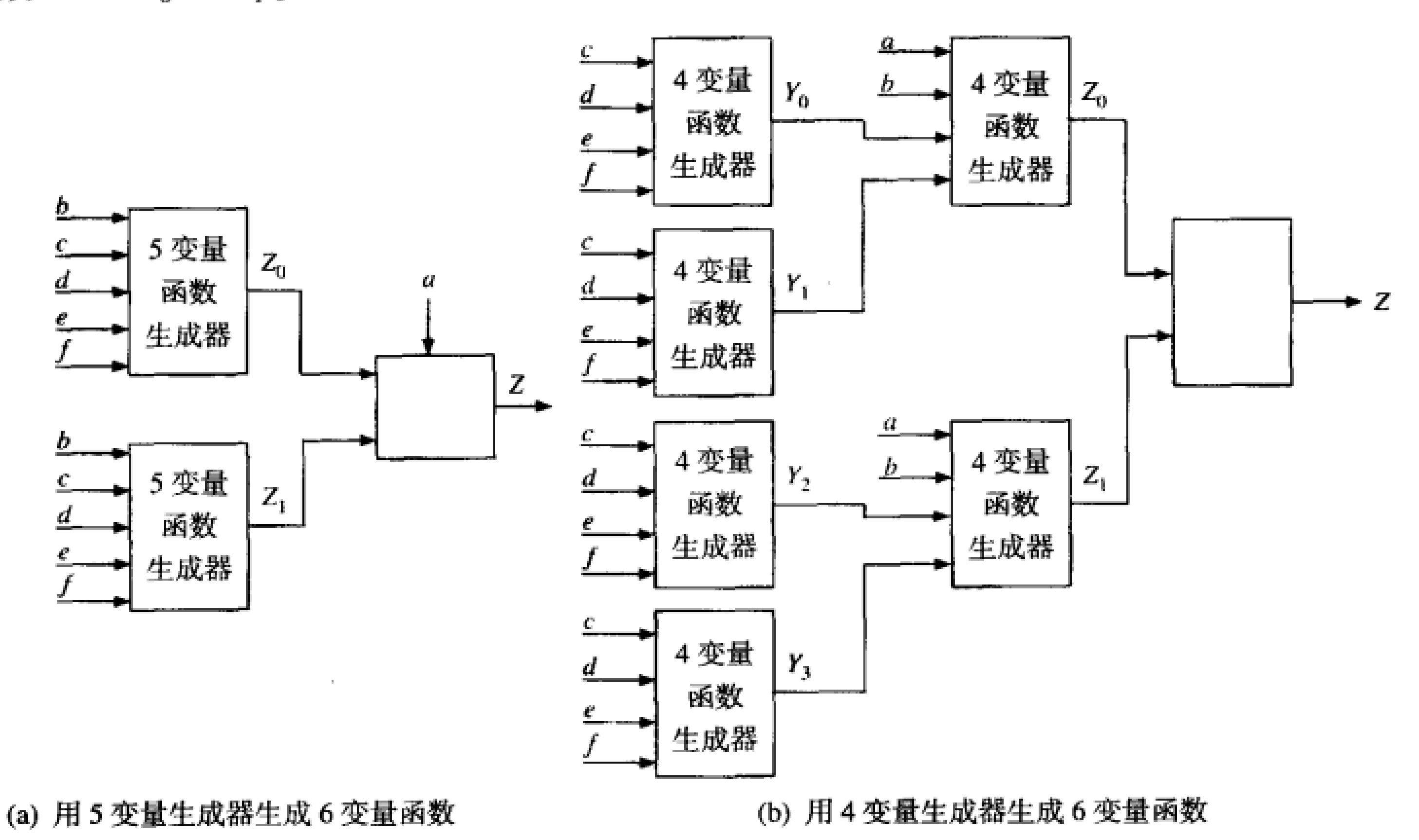
香农展开定理可以把一个具有很多变量的函数分解为几个具有较少变量的函数。上一节中，我们把一个 4 选 1 多路选择器分解为几个 2 选 1 多路选择器，以便可以在具有 4 变量函数生成器的逻辑模块中实现。香农展开定理为分解任意函数提供了一个通用的技术。

下面我们通过分解一个任意 6 变量函数 $Z(a,b,c,d,e,f)$ 来说明香农分解。首先，展开函数如下：

$$Z(a,b,c,d,e,f) = a'Z(0,b,c,d,e,f) + aZ(1,b,c,d,e,f) = a'Z_0 + aZ_1 \tag{6.3}$$

我们可以通过如下方法验证式(6.3)的正确性：首先设 a 在等式两边均为 0；接着设 a 在两边

为 1，分别观察等式是否成立。因为当 $a=0$ 和 $a=1$ 时等式均成立，所以式(6.3)是正确的。式(6.3)对应的电路见图 6.6(a)，电路使用了两个单元来实现 Z_0 和 Z_1，并用第三个单元的一半来实现 3 变量函数 $Z=a'Z_0+aZ_1$。



(a) 用 5 变量生成器生成 6 变量函数　　(b) 用 4 变量生成器生成 6 变量函数

图 6.6　6 变量函数的实现

作为一个例子，考虑下面的函数：

$$Z=abcd'ef'+a'b'c'def'+b'cde'f$$

设 $a=0$，则有

$$Z=0\cdot bcd'ef'+1\cdot b'c'def'+b'cde'f=b'c'def'+b'cde'f$$

设 $a=1$，则有

$$Z=1\cdot bcd'ef'+0\cdot b'c'def'+b'cde'f=bcd'ef'+b'cde'f$$

由于 Z_0 和 Z_1 均为 5 变量函数，所以它们每一个都需要一个 5 输入 LUT。一个函数不管有多少个乘积项，只要是 5 变量函数，就可以用一个 5 输入 LUT 来实现。我们还需要一个 2 选 1 多路选择器或是另一个 LUT5，从 Z_0 和 Z_1 生成 Z。

如果只有 4 变量 LUT 可用，那么应把 5 变量函数再分解成两个 4 变量函数。为此，可以通过两次使用香农展开定理（第一次对 a 展开，第二次对 b 展开）来实现，或者也可以通过分解为 4 个函数来一步实现，如下所示：

$$\begin{aligned}Z(a,b,c,d,e,f)&=a'b'\cdot Z(0,0,c,d,e,f)+a'b\cdot Z(0,1,c,d,e,f)+\\&\quad ab'\cdot Z(1,0,c,d,e,f)+ab\cdot Z(1,1,c,d,e,f)\\&=a'b'\cdot Y_0+a'b\cdot Y_1+ab'Y_2+ab\cdot Y_3\end{aligned}\tag{6.4}$$

图 6.6(b)说明了基于 4 变量函数实现一个 6 变量函数的一般原理。

现在我们考虑把函数

$$Z=abcd'ef'+a'b'c'def'+b'cde'f$$

分解为 4 变量函数。对 a 和 b 应用香农展开定理。

- 代入 $a=b=0$，得 $Y_0=c'def'+cde'f$。
- 代入 $a=0,b=1$，得 $Y_1=0$。
- 代入 $a=1,b=0$，得 $Y_2=cde'f$。
- 代入 $a=b=1$，得 $Y_3=cd'ef'$。

在一般的函数实现中，7 个 4 变量函数生成器可以实现一个 6 变量函数，如图 6.6(b)所示。但是，在本例中，展开得到的 4 变量函数中有一个是零函数，函数进一步化简为

$$Z=a'b'\cdot Y_0+ab'\cdot Y_2+ab\cdot Y_3$$

因此，5 个 4 变量函数生成器就足以实现这一函数，其中 3 个实现 Y_0, Y_2 和 Y_3，再一个实现 $Z_1=ab'Y_2+ab\cdot Y_3$，最后一个实现 $Z=a'b'\cdot Y_0+Z_1$。图 6.7 说明了基于 4 变量函数生成器实现函数 $Z=abcd'ef'+a'b'c'def'+b'cde'f$ 的原理。

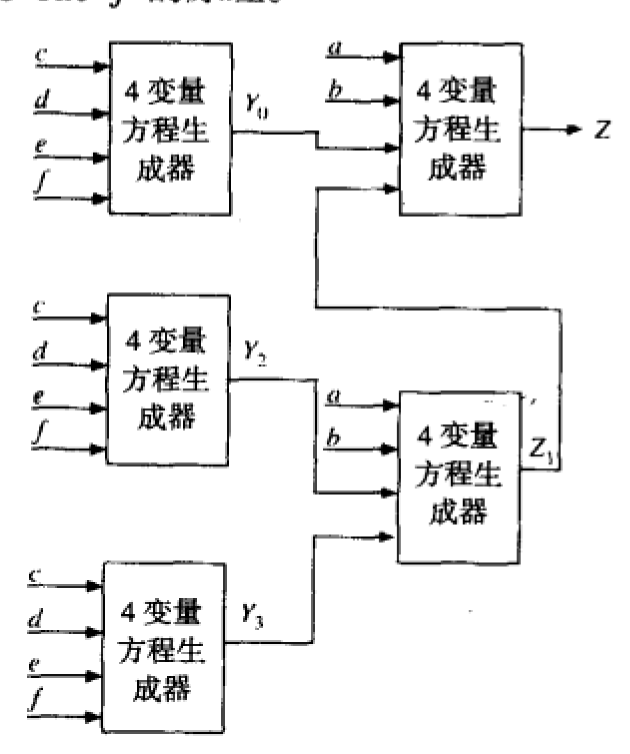


图 6.7 使用 4 变量函数生成器实现函数

任意 7 变量函数均可以用 6 个或更少的 LUT5 来实现。任何一个 7 变量函数的展开式为

$$\begin{aligned}Z(a,b,c,d,e,f,g)&=a'b'\cdot Z(0,0,c,d,e,f,g)+a'b\cdot Z(0,1,c,d,e,f,g)+\\&\quad ab'\cdot Z(1,0,c,d,e,f,g)+ab\cdot Z(1,1,c,d,e,f,g)\\&=a'b'\cdot Y_0+a'b\cdot Y_1+ab'\cdot Y_2+ab\cdot Y_3\end{aligned}\tag{6.5}$$

这里，Y_0, Y_1, Y_2, Y_3 均为 c, d, e, d 和 g 的 5 变量函数。通过两次应用香农展开定理（第一次对 a 展开，第二次对 b 展开），可以得到等式(6.5)。作为一个例子，考虑 7 变量函数：

$$Z=c'de'fg+bcd'e'fg'+a'c'def'g+a'b'd'ef'g'+ab'defg'$$

- 代入 $a=b=0$，得 $Y_0=c'de'fg+c'def'g+d'ef'g'$。
- 代入 $a=0,b=1$，得 $Y_1=c'de'fg+cd'e'fg'+c'def'g$。
- 代入 $a=1,b=0$，得 $Y_2=c'de'fg+defg'$。
- 代入 $a=b=1$，得 $Y_3=c'de'fg+cd'e'fg'$。

这一函数可以使用 6 个 5 变量函数生成器来实现。其中 4 个生成器实现 Y_0, Y_1, Y_2 和 Y_3，再一个生成器实现 4 变量函数 $Z_0=ab'\cdot Y_0+ab\cdot Y_1$，最后一个实现 5 变量函数 $Z=Z_0+ab'\cdot Y_2+ab\cdot Y_3$。

我们使用香农分解，可以把一个 n 变量函数分解为两个 n–1 变量函数和几个多路选择器。在

本章的前几节中，我们已看到用 LUT 实现多路选择器是非常低效的。当函数的变量数(n)增加时，实现 n 变量函数所需的 LUT 的数量也随之快速增加。如果使用多路选择器则可以大幅度降低 LUT 表的使用数量。基于这个原因，有些 FPGA 除了提供 LUT4 外，还提供多路选择器。

例　使用 4 输入 LUT 和 2 选 1 多路选择器实现一个 7 变量函数。

解：由香农展开定理得

7 变量函数生成器=2 个 6 变量函数生成器+1 个 2 选 1 多路选择器　(i)

6 变量函数生成器=2 个 5 变量函数生成器+1 个 2 选 1 多路选择器　(ii)

5 变量函数生成器=2 个 4 变量函数生成器+1 个 2 选 1 多路选择器　(iii)

把(iii)代入(ii)，得

6 变量函数生成器=4 个 4 变量函数生成器+3 个 2 选 1 多路选择器　(iv)

把(iv)代入(i)，得

7 变量函数生成器=8 个 4 变量函数生成器+7 个 2 选 1 多路选择器

所以，一个 7 变量函数的具体实现见图 6.8 所示。

如果只有 4 输入 LUT 可用，那么 7 变量函数需要 15 个 4 输入 LUT。一个 2 选 1 多路选择器比一个 4 输入 LUT 便宜，因此 7 变量函数可以用 8 个 4 输入 LUT 和 7 个 2 选 1 多路选择器实现，如图 6.8 所示。

Xilinx Spartan 生产的 FPGA 中既提供了 4 输入 LUT，也提供了多路选择器。这种 FPGA 的基本逻辑单元称为逻辑片（slice），可以用图 6.9 简单表示。每个逻辑片内部包含两个 4 输入 LUT 和两个 2 选 1 多路选择器（还有一些其他的逻辑在这里没有画出）。一个 7 变量函数可以用 4 个这样的逻辑片实现，如图 6.10 所示。点画线表示每个逻辑片。

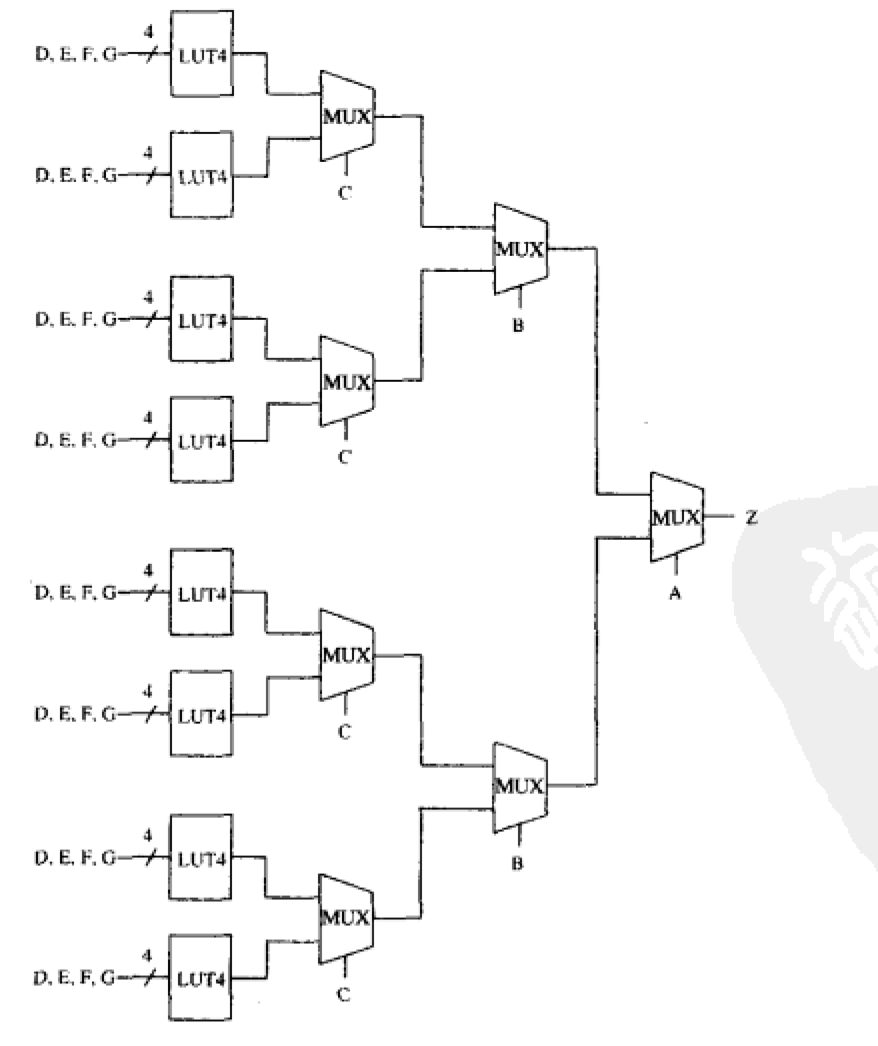


图 6.8　使用 4 输入 LUT 和 2 选 1 MUX 实现 7 变量函数

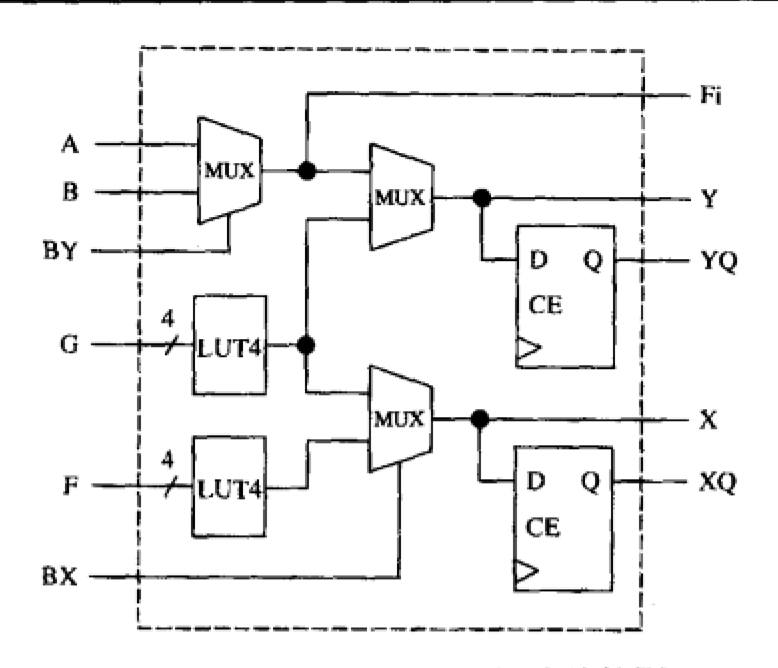


图 6.9 Xilinx Spartan 逻辑片的简图

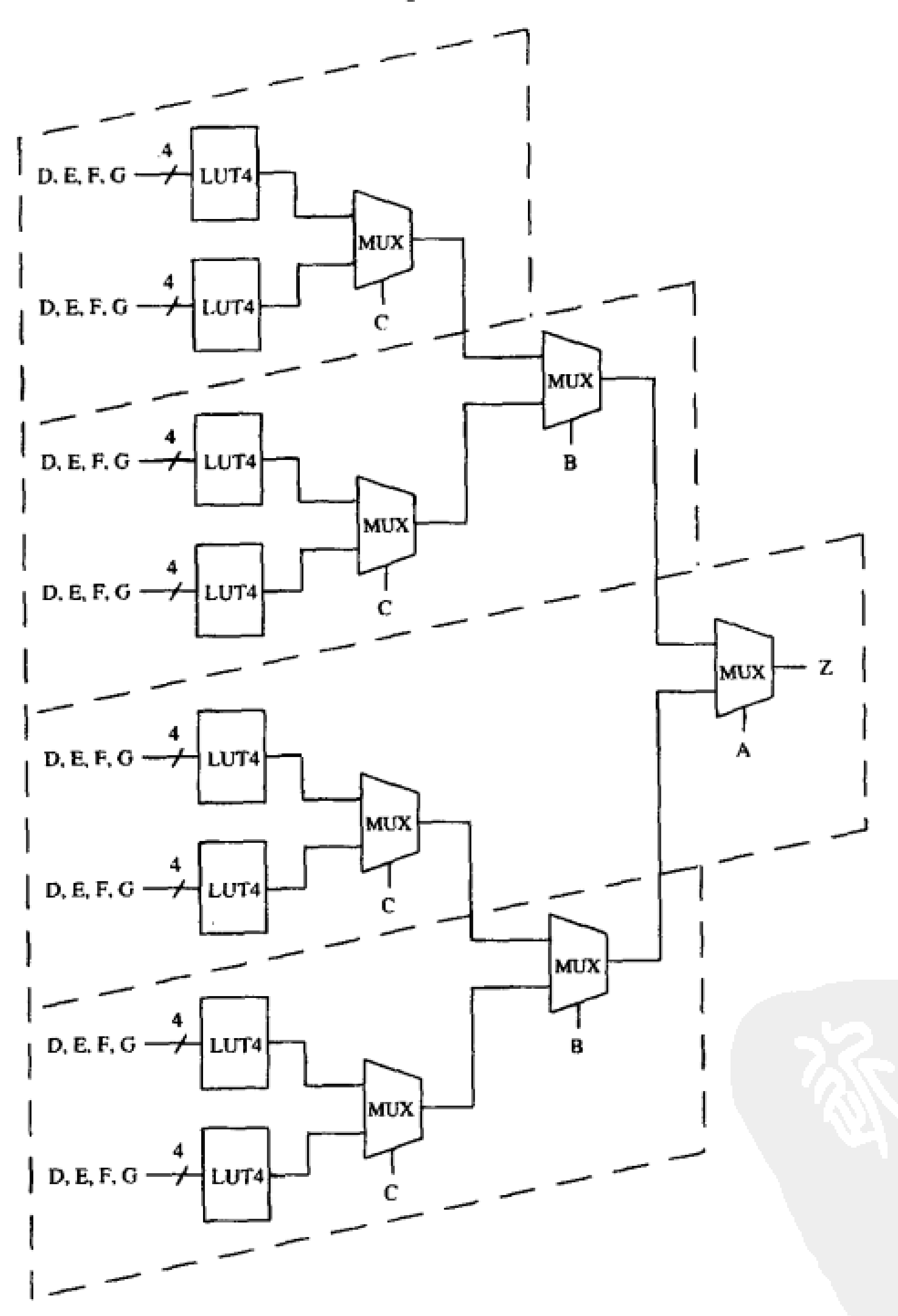


图 6.10 使用 4 个 Xilinx Spartan 逻辑片实现一个 7 变量函数

作为另一个例子，用 4 变量函数生成器实现一个奇偶校验函数。奇偶校验函数为

$$F = A \oplus B \oplus C \oplus D \oplus E$$

如果把上式展开为与或表达式，那么一共有 16 个乘积项，但它是 5 变量函数。由香农展开

定理可知，任何 5 变量函数均可以分解为 2 个 4 变量函数，并且可以用 2 个 4 输入 LUT 和 1 个 2 选 1 多路选择器来实现。对于这一特殊函数来说，2 个 4 变量函数生成器就足以实现，因为该函数可以分解为一个 4 变量奇偶校验函数与第五个变量的异或形式。

6.3　FPGA 的进位链

用 FPGA 实现加法器的最简单方法是用 FPGA 逻辑模块计算每一位的和与进位。可以用一个 4 变量 LUT（现在已经是标准构建模块了）生成 *sum* 表达式，另一个 LUT4 用来实现进位表达式。每个进位都通过互连资源传递到下一位求和运算中。然而，由于加法运算是一种基本和通用运算，很多 FPGA 都提供生成和传递进位的专门电路。通常，进位可以通过专用进位链来实现。作为一个例子，考虑示于图 6.11 的进位链。每个 LUT 产生相应输入位的和（*a*, *b* 和 *Carry-in*）。进位链并行生成进位，并通过专用互连把它们分别连接到下一位加法运算的 LUT 上。

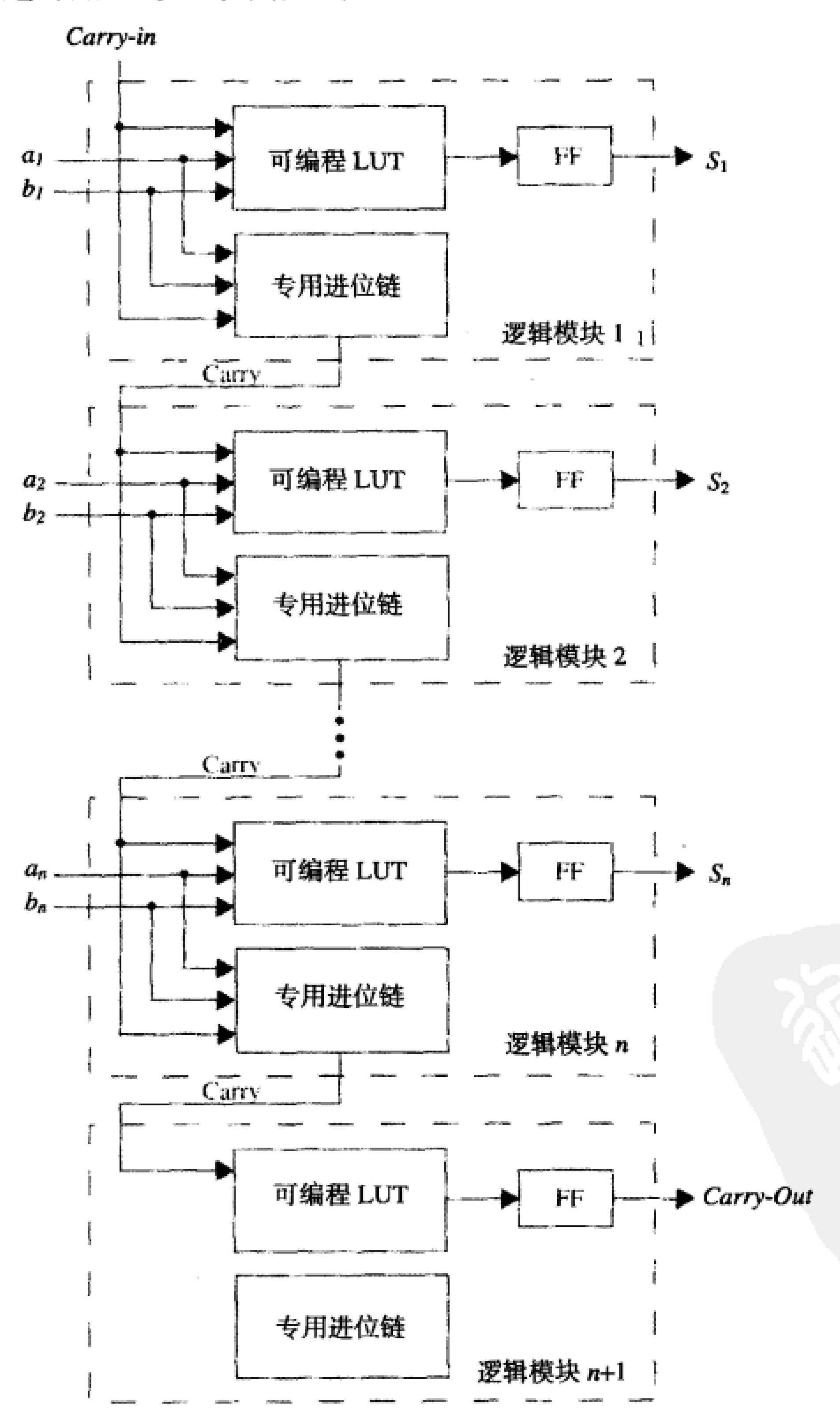


图 6.11　快速加法运算中的进位链

如果没有这样的进位链，那么实现一个 n 位加法器通常需要使用 $2n$ 个逻辑模块（若逻辑模块为 LUT4）。现在，通过使用进位链，我们只需使用 n 个逻辑模块（尽管需要一些其他的专用电路）。专用电路生成进位并把它传递到下一个 LUT4，所以在很多电路中我们都不用硬件来实现进位生成器。但是由于加法运算是很通用普遍的，所以如果在 FPGA 中包含它的实现逻辑模块也是有必要的。

6.4 FPGA 中的级联链

有些 FPGA 支持把多个 FPGA 模块的输出级联起来。最常用的级联类型为与（AND）配置和或（OR）配置。通常我们不使用单独的函数生成模块来完成逻辑模块输出的与运算（或者或运算），而是把逻辑模块的输出直接连到级联链电路以实现与运算（或者或运算）。图 6.12 给出了一个使用 4 输入 LUT 实现函数的 FPGA 级联链的例子。因此，如果要实现一个 32 变量的或运算，那么我们可以使用 8 个逻辑模块。每个逻辑模块均生成一个 4 变量或运算，再把之前的逻辑模块的输出通过级联的或门或起来即可。有些 FPGA 中也可以提供级联与门和异或门。在基于 LUT 的 FPGA 中，这种级联可以称为 LUT 链。

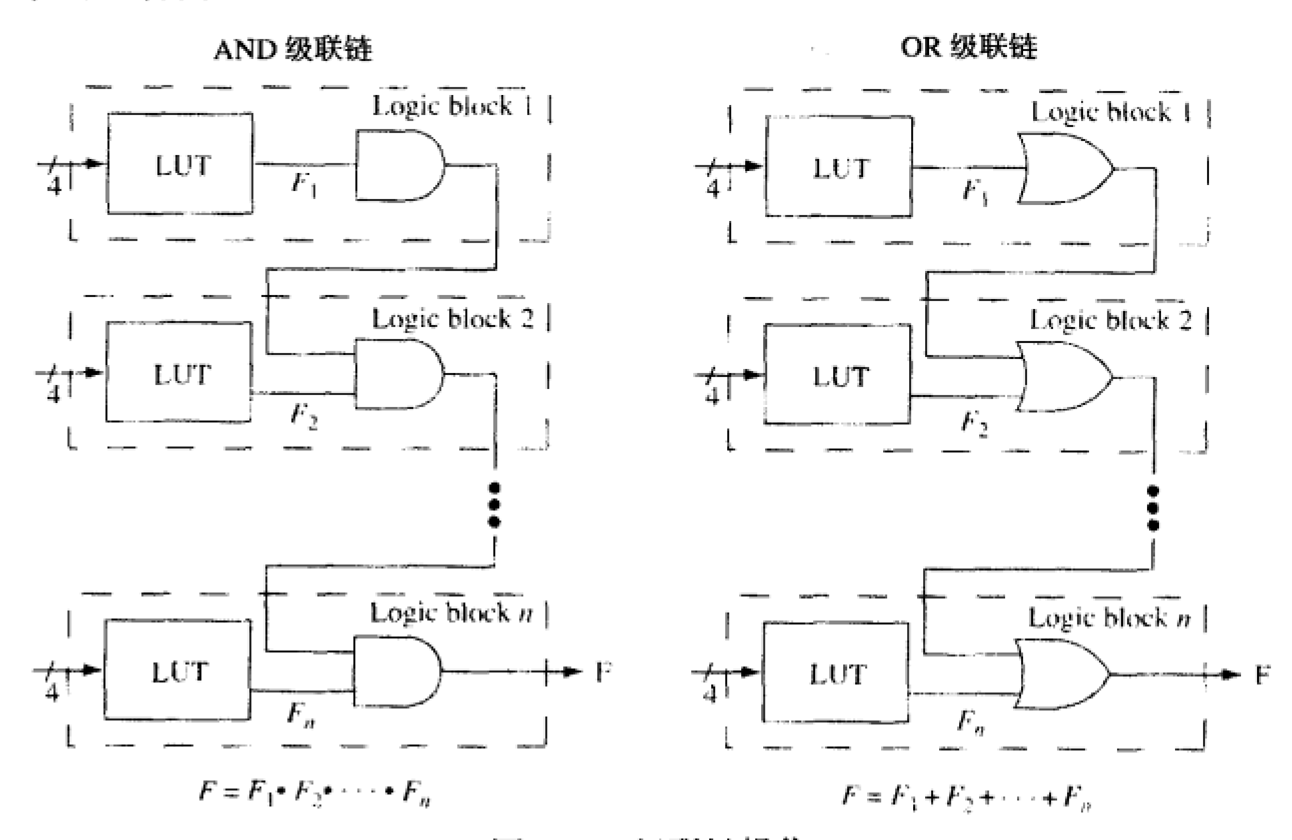


图 6.12 级联链操作

6.5 商用 FPGA 的逻辑模块举例

我们将介绍三种商用 FPGA 的逻辑模块，它们分别由 Xilinx, Altera 和 Actel 公司生产。Xilinx 和 Altera 生产的 FPGA 都是用 4 变量 LUT 作为基本构建模块，Actel 生产的 FPGA 使用多路选择器和门电路作为基本模块。

6.5.1 Xilinx 可配置逻辑模块

Xilinx 生产的 Spartan 和 Virtex 家族 FPGA 通常使用 2 ~ 4 个基本模块（称为 slice）构成一个可配置逻辑模块（CLB），如图 6.13 所示。CLB 是 Xilinx 生产的 FPGA 中可编程逻辑模块的名称术语。每个逻辑片包含两个函数生成器：G 函数生成器和 F 函数生成器。另外，还含有两个多路选择器（F5 和 FX）用以函数实现。为了实现 4 变量 LUT，需要使用 16 位 SRAM。因此，每个逻辑片中含有 32 位 SRAM 以生成组合逻辑函数。F5 多路选择器可以用于组合两个 4 变量函数生

成器的输出以形成一个 5 变量函数生成器。多路选择器的选择输入端也可以用做第 5 个输入变量。FX 多路选择器的所有输入均可以使用，因此我们可以生成几种 2 变量函数。这一多路选择器与两个逻辑片的选择输出 F5 结合起来可以形成一个 6 输入函数。每个逻辑片还含有两个触发器，它们既可以作为边沿触发的 *D* 触发器，也可以作为电平触发的锁存器。该逻辑片支持加法的快速进位生成，除了一般 4 变量 LUT 之外，还含有一些其他的逻辑以便实现一些特定的逻辑函数。

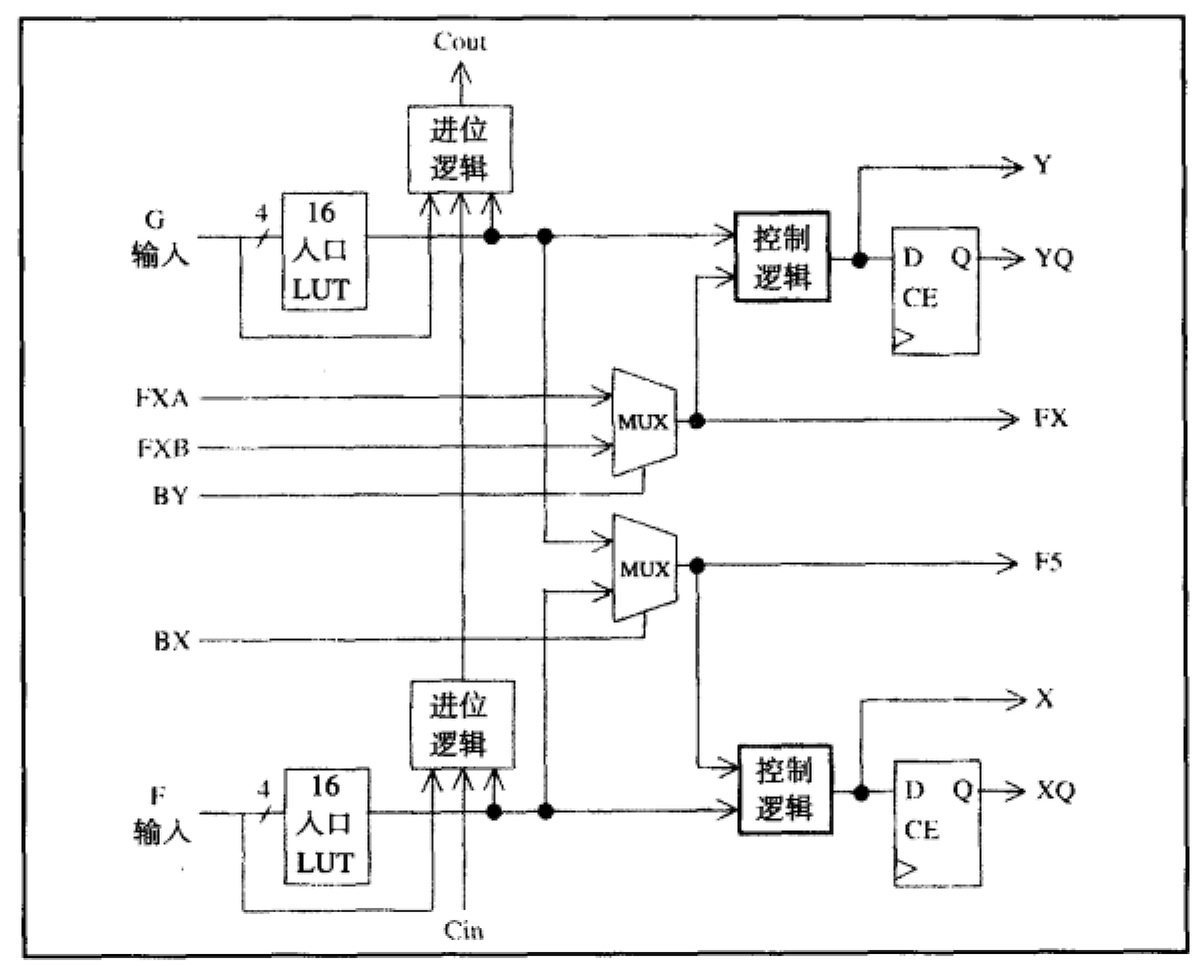


图 6.13 Xilinx Spartan 逻辑片和 Virtex 逻辑片简图

6.5.2 Altera 逻辑单元

Altera 公司将其生产的基本逻辑模块命名为逻辑单元(LE)。图 6.14 给出了 Altera Stratix FPGA 的 LE 略图。每个 LE 都包含一个 4 变量 LUT 和一个触发器，可以实现任意 4 变量函数。LE 的输出既可以来自于组合逻辑电路，也可以来自于触发器。几个相邻的 LE 可以通过级联链连接起来，所以 LE 也可以实现多于 4 变量的函数。LE 中还包含用于加法计算的快速进位链。触发器可以同步清零或置位。由于这是一个简单的示意图，所以很多具体电路都被省略了。触发器的输出可以反馈回 LUT 的输入。该逻辑单元还有一些逻辑门对 LUT 的输入可以进行操作。

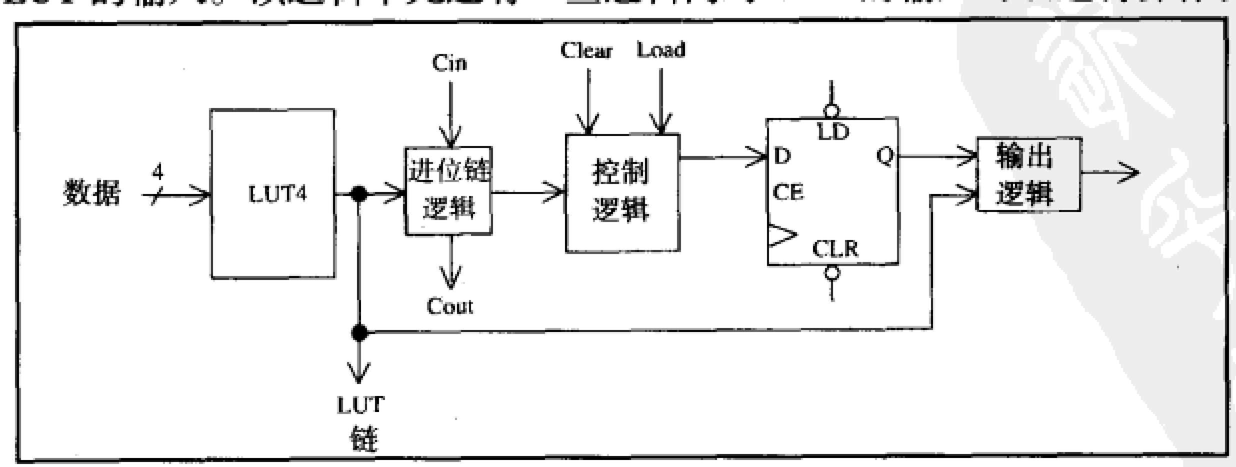


图 6.14 Altera Stratix 逻辑单元简图

6.5.3 Actel Fusion 逻辑单元

Actel 公司生产的 FPGA 的基本构建单元称为 VersaTile，它是由多个多路选择器和门电路构成的，如图 6.15 所示。VersaTile 模块有四个输入：X_1, X_2, X_3 和 X_c。每个 VersaTile 都可以做如下任意一个配置。

- 一个三输入逻辑函数。
- 一个具有清零和置位端的锁存器。
- 一个具有清零和置位端的 D 触发器。
- 一个具有清零、置位和使能端的 D 触发器。

当用做一个三输入逻辑函数时，输入为 X_1, X_2 和 X_3；当用做锁存器和触发器时，输入 X_2 通常用做时钟输入端，输入 X_1 和 X_c 用做触发器的使能端和清零端。逻辑模块为快速本地链接和有效长线链接都提供了双重输出，但是为了简单起见，在图 6.15 中我们只画了一个输出。VersaTile 的晶粒细化比其他 FPGA 中的 4 输入 LUT 高很多，与标准门阵列（即传统的掩模可编程逻辑阵列）相当。

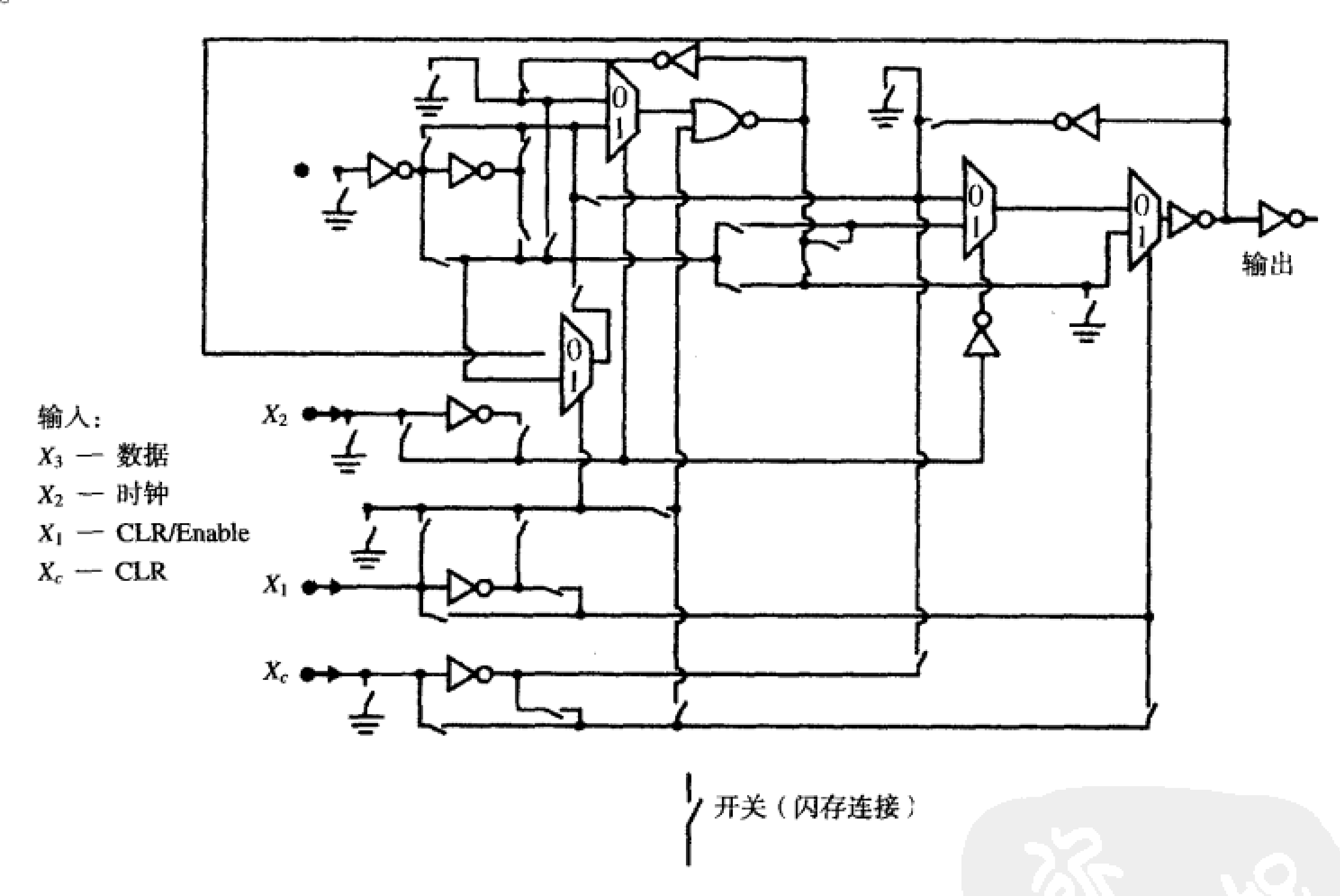


图 6.15 Actel Fusion 和 ProAS1C 逻辑模块简图

6.6 FPGA 中的专用存储器

在许多实际应用电路中都用到了存储器。存储器可以存放一个常数表用做处理系数，也可以用做你要设计的 FPGA 嵌入式处理器的指令或数据的内存。早期的 FPGA 都不含有专用存储器，当需要使用存储器时，设计者一般通过接口连接 FPGA 和外部存储器芯片。随着芯片集成度的提高，FPGA 的设计者开始把专用存储器嵌入到 FPGA 芯片中，省去了与外部存储器的接口问题。

现代 FPGA 中内含 16 kb ~ 10 Mb 比特的专用存储器。表 6.1 给出了一些 FPGA 中的专用 RAM。例如 Xilinx Virtex-5 内含 1 ~ 10 Mb 专用存储器；Altera StratixII 内含 409 kb ~ 9 Mb 存储器；Actel Fusion 内含 27 ~ 270 kb 存储器。专用存储器一般用 FPGA 中的一些（4 ~ 1000）大的专用 SRAM 模块来实现。专用 RAM 模块的典型布局结构示于图 6.16。在很多 FPGA 中，专用 RAM 模块配置在逻辑模块阵列的外围（如 Xilinx Virtex/Spartan 和 Actel Fusion）；在另一些 FPGA 中，专用 RAM 模块是以存储器列的形式分布在 FPGA 的几个区域(如 Altera Stratix)。在很多 FPGA 中，SRAM 模块的大小相同(如 Xilinx Virtex 中为 18 kb)；在有些 FPGA 中，SRAM 模块的大小不同，例如 Altera StratixII 中含有 521 b, 4 kb 和 512 kb 模块。Xilinx 的专用存储器称为块 RAM(block RAM)，Altera 的专用存储器称为 TriMatrix 存储器。有些 FPGA 在 SRAM 中提供奇偶校验位。有些生产厂家在计算 RAM 大小时是包含奇偶校验位在内的，而有些厂家在计算 RAM 大小时，只是计算有效存储单元而不包括奇偶校验位。

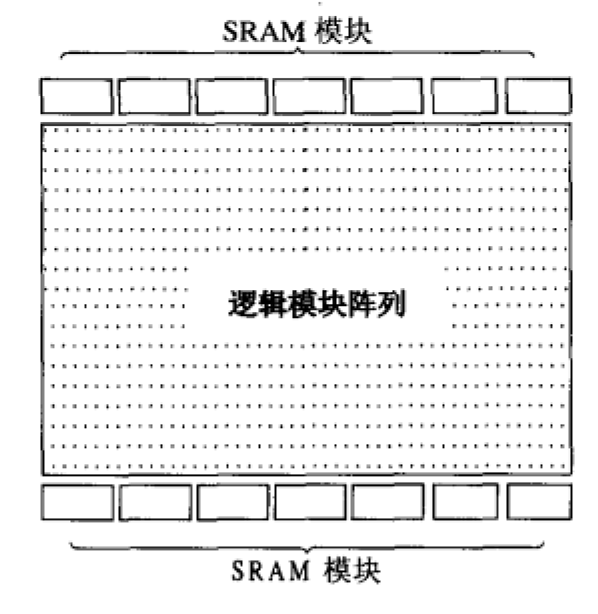


图 6.16　FPGA 中的内嵌 RAM

现代 FPGA 中的专用 RAM，其主要特点是提供多个可供选择的 RAM 宽度。如表 6.1 所示，现在有很多种存储器模块，它们可以按不同方式排列以得到不同的存储大小。假设一个 RAM 模块可以提供 32 kb 的 SRAM，那么它可以用做 32 K×1, 16 K×2, 8 K×4 和 4 K×8 存储器。这样，RAM 的宽度可以根据具体的应用来选择，在一个应用中可能用到 8 字节宽的存储器，而在另一应用中也可能用到 64 比特宽的存储器。

表 6.1　FPGA 中专用 RAM 的大小

FPGA 系列	专用 RAM 大小/kb	组　织
Xilinx Virtex 5	1152 ~ 10 368	64 ~ 576　18 kb 模块
Xilinx Virtex 4	864 ~ 9936	48 ~ 552　18 kb 模块
Xilinx Virtex-II	72 ~ 3024	4 ~ 168　18 kb 模块
Xilinx Spartan 3E	72 ~ 648	4 ~ 36　18 kb 模块
Altera Stratix II	409 ~ 9163	104 ~ 930　512 b 模块
		78 ~ 768　4 kb 模块
		0 ~ 9　512 kb 模块
Altera Cyclone II	117 ~ 1125	26 ~ 250　4 kb 模块
Lattice SC	1054 ~ 7987	56 ~ 424　18 kb 模块
Actel Fusion	27 ~ 270	6 ~ 60　4 kb 模块

基于 LUT 的 FPGA 提供另一种可选存储器。如果设计时只需要很小的存储器，那么就可以用 LUT 实现存储功能，而不用使用专用存储器。众所周知，一个 4 变量 LUT 包含 16 比特存储单元。我们可以通过把几个 LUT 的存储单元连接起来构成一个较小的存储器。2 个 4 输入 LUT（如图 6.17 所示）可以构成一个 32×1 或 16×2 存储器。当使用 32×1 存储器时，必须要使用 5 条地址线和 1 条数据线（D_1 和 D_2 必须连接起来）。在地址线 MSB 为 0 时，上边的 LUT 必须有效；当地址线最高有效位为 1 时，下边的 LUT 必须有效。这可以通过控制地址线的最高位和反相器实现。当使用 16×2 存储器时，两个 LUT 同时被选，且数据线 D1 和 D2 并行输出。由 LUT 单元构成的存储器称为分布式存储器

（Xilinx 术语）。顾名思义，这种存储器是分布在芯片各个逻辑模块里的。分布式存储器的缺点在于一旦使用了 LUT 存储器，则该逻辑模块一般不可用。LUT 存储器可以用做异步存储器，也可以同逻辑模块触发器一起构成同步存储器。表 6.2 给出了一些 FPGA 基于 LUT 的存储器大小。

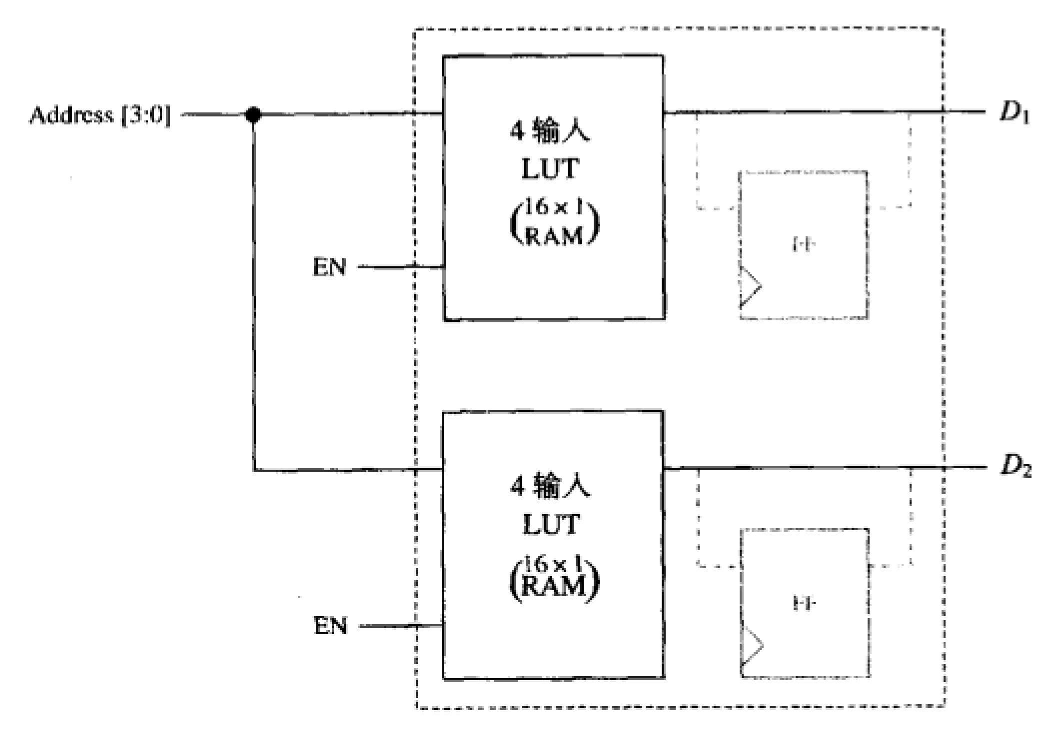


图 6.17 用 LUT 创建内存

表 6.2 FPGA 中基于 LUT 的 RAM

FPGA 系列	基于 LUT 的 RAM /kb	LUT 个数
Xilinx Virtex 5	320 ~ 3420	19 200 ~ 207 360
Xilinx Virtex 4	96 ~ 987	12 288 ~ 1 126 336
Xilinx Virtex-II	8 ~ 1456	512 ~ 93 184
Xilinx Spartan 3E	15 ~ 231*	1920 ~ 29 504
Altera Stratix II	195 ~ 2242**	12 480 ~ 143 520
Altera Cyclone II	72 ~ 1069**	4608 ~ 68 416
Lattice SC	245 ~ 1884	15 200 ~ 115 200
Lattice ECP2	12 ~ 136	6000 ~ 68 000

* 所有的 LUT 并没有都用做分布式 RAM。

** 根据 LUT 个数计算。

6.6.1 FPGA 存储器的 VHDL 模型

FPGA 的内嵌存储器可以通过行为描述 VHDL 模块来实现。存储器可以是同步的，也可以是异步的。异步存储器的读操作是指数据在存取时间之后出现在输出总线上，与时钟无关。相反，同步存储器的读写控制线只受有效时钟控制。有些存储器中写操作是同步的，而读操作是异步的。

现代综合工具能在高层面（行为描述）上实现内嵌存储器。图 6.18 所示的 VHDL 程序实现了一个写同步、读异步存储器。存储器阵列由无符号矢量表示。由于 Address 的数据类型为无符号矢量，所以在标识存储器阵列时，它必须转化为整数。因此，我们使用 IEEE numeric_bit 库中的转换函数来实现这一目的。写操作在进程语句中实现，并且进程语句只在时钟上升沿到来时有效；读操作在进程语句外实现，因此它的实现与时钟无关。用现在的 Xilinx 工具对该代码进行综合就会得到分布式存储器。对于异步存储器来说，由于 LUT 产生的输出不同步，所以分布式存储

是很理想的。相反，图 6.19 代码却实现块 RAM。在这一代码中，读操作是在进程语句中进行的，所以读操作与时钟沿同步。

```
library IEEE;
use IEEE.numeric_bit.all;

entity Memory is
  port(Address: in unsigned(6 downto 0);
       CLK, MemWrite: in bit;
       Data_In: in unsigned(31 downto 0);
       Data_Out: out unsigned(31 downto 0));
end Memory;

architecture Behavioral of Memory is
type RAM is array (0 to 127) of unsigned(31 downto 0);
signal DataMEM: RAM;  -- no initial values
begin
  process(CLK)
  begin
    if CLK'event and CLK = '1' then
      if MemWrite = '1' then
        DataMEM(to_integer(Address)) <= Data_In;  -- Synchronous Write
      end if;
    end if;
  end process;

  Data_Out <= DataMEM(to_integer(Address));  -- Asynchronous Read
end Behavioral;
```

图 6.18　实现 LUT 存储器的典型行为描述 VHDL 代码

```
library IEEE;
use IEEE.numeric_bit.all;

entity Memory is
  port(Address: in unsigned(6 downto 0);
       CLK, MemWrite: in bit;
       Data_In: in unsigned(31 downto 0);
       Data_Out: out unsigned(31 downto 0));
end Memory;

architecture Behavioral of Memory is
type RAM is array (0 to 127) of unsigned(31 downto 0);
signal DataMEM: RAM;  -- no initial values
begin
  process(CLK)
  begin
    if CLK'event and CLK = '1' then
      if MemWrite = '1' then
        DataMEM(to_integer(Address)) <= Data_In;  -- Synchronous Write
      end if;
      Data_Out <= DataMEM(to_integer(Address));   -- Synchronous Read
    end if;
  end process;
end Behavioral;
```

图 6.19　实现专用存储器的典型行为描述 VHDL 代码

如果基于 ROM 法（查表法）来实现电路，那么为了实现 LUT，综合工具可能生成 RAM。作为一个例子，考虑实现一个基于 LUT 法的 4×4 乘法器，其 VHDL 代码示于图 6.20。因为使用 LUT 法，所以与每个输入组合对应的乘积都存储在 LUT 中。由于被乘数和乘数均为 4 位，那么一共有 256 中可能的输入组合。我们使用一个常数数组存储所有的乘积。由于被乘数开始为 0000，所以头 16 个积均为 0；下一个被乘数为 0001，由于乘数为 0 ~ 15，所以下 16 个积为 0 ~ 15（十进制）。这一乘法器的 VHDL 代码示于图 6.20。如果这一代码用现在的 Xilinx 工具综合就生成分布式 RAM 以存储这些乘积数。分布式 RAM 的读操作是异步的，因为逻辑模块中的 LUT 输出一直随着输入而改变，不需要时钟信号。然而，当不想浪费 LUT 去实现存储器时，就把数组存储在专用块 RAM 中。如果读操作是同步的，比如：

```
process(CLK)
begin
      if CLK'event and CLK = '1' then
      Product <= PROD_ROM(to_integer(Mplier & Mcand));
              -- read Product LUT (Synchronously)
      end if;
end process;
```

则当前 Xilinx 综合工具可以生成专用块 RAM 来存储这 256 个乘数积。

```
library IEEE;
use IEEE.numeric_bit.all;

entity LUTmult is
  port(Mplier, Mcand: in unsigned(3 downto 0);
       Product: out unsigned(7 downto 0));
end LUTmult;

architecture ROM1 of LUTmult is
type ROM is array (0 to 255) of unsigned(7 downto 0);
constant PROD_ROM: ROM: =
     (x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00",
     x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07", x"08", x"09", x"0A", x"0B", x"0C", x"0D", x"0E", x"0F",
     x"00", x"02", x"04", x"06", x"08", x"0A", x"0C", x"0E", x"10", x"12", x"14", x"16", x"18", x"1A", x"1C", x"1E",
     x"00", x"03", x"06", x"09", x"0C", x"0F", x"12", x"15", x"18", x"1B", x"1E", x"21", x"24", x"27", x"2A", x"2D",
     x"00", x"04", x"08", x"0C", x"10", x"14", x"18", x"1C", x"20", x"24", x"28", x"2C", x"30", x"34", x"38", x"3C",
     x"00", x"05", x"0A", x"0F", x"14", x"19", x"1E", x"23", x"28", x"2D", x"32", x"37", x"3C", x"41", x"46", x"4B",
     x"00", x"06", x"0C", x"12", x"18", x"1E", x"24", x"2A", x"30", x"36", x"3C", x"42", x"48", x"4E", x"54", x"5A",
     x"00", x"07", x"0E", x"15", x"1C", x"23", x"2A", x"31", x"38", x"3F", x"46", x"4D", x"54", x"5B", x"62", x"69",
     x"00", x"08", x"10", x"18", x"20", x"28", x"30", x"38", x"40", x"48", x"50", x"58", x"60", x"68", x"70", x"78",
     x"00", x"09", x"12", x"1B", x"24", x"2D", x"36", x"3F", x"48", x"51", x"5A", x"63", x"6C", x"75", x"7E", x"87",
     x"00", x"0A", x"14", x"1E", x"28", x"32", x"3C", x"46", x"50", x"5A", x"64", x"6E", x"78", x"82", x"8C", x"96",
     x"00", x"0B", x"16", x"21", x"2C", x"37", x"42", x"4D", x"58", x"63", x"6E", x"79", x"84", x"8F", x"9A", x"A5",
     x"00", x"0C", x"18", x"24", x"30", x"3C", x"48", x"54", x"60", x"6C", x"78", x"84", x"90", x"9C", x"A8", x"B4",
     x"00", x"0D", x"1A", x"27", x"34", x"41", x"4E", x"5B", x"68", x"75", x"82", x"8F", x"9C", x"A9", x"B6", x"C3",
     x"00", x"0E", x"1C", x"2A", x"38", x"46", x"54", x"62", x"70", x"7E", x"8C", x"9A", x"A8", x"B6", x"C4", x"D2",
     x"00", x"0F", x"1E", x"2D", x"3C", x"4B", x"5A", x"69", x"78", x"87", x"96", x"A5", x"B4", x"C3", x"D2", x"E1");

begin
Product <= PROD_ROM(to_integer(Mplier&Mcand));  -- read Product LUT
end ROM1;
```

图 6.20　基于 LUT 的 4×4 乘法器

6.7 FPGA 中的专用乘法器

很多现代 FPGA 中都提供专用乘法器。假设设计者需要一个 16×16 的乘法器。如果没有专用乘法器，那么将会用多个可编程逻辑模块来实现此 16×16 乘法器。这个乘法器从所用的逻辑模块数量和互连资源来看代价很高。同时，由于连接乘法器的各部分需要使用开关，该乘法器速度也慢。专用乘法器比用逻辑模块实现的乘法器，不仅节省面积，而且速度也更快。由于乘法操作是很多应用实例中的重要组成部分，所以很多商用 FPGA 中都提供专用乘法器。例如，Xilinx Virtex4/Spartan-3 和 Altera Stratix/Cyclone 生产的 FPGA 均包含 18×18 乘法器。这些乘法器可以计算两个 18 位操作数的乘法，得到一个 36 位的积，如图 6.21 所示。乘数和被乘数都可以置入可选寄存器中，积也可以置入一个可选积寄存器中。乘法器的输入既可以来自外部管脚，也可以来自 FPGA 中的其他逻辑模块。

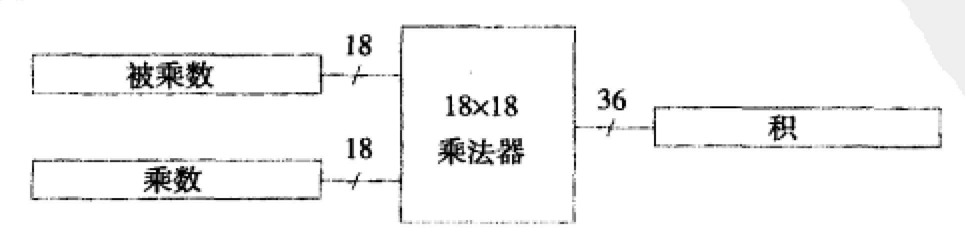


图 6.21　专用乘法器

当我们需要计算两个多于 18 位数的乘法时，可以把多个内嵌专用乘法器连在一起用。若 A 和 B 均为 32 位，设 C, D, E 和 F 均为 16 位，并有

$$A = C \times 2^{16} + D$$

$$B = E \times 2^{16} + F$$

则有 $AB = CE \times 2^{32} + (DE + CF) \times 2^{16} + DF$ 。这就意味着用 4 个乘法器生成部分积 CE, DE, CF 和 DE，再需要几个加法器把这些部分积加在一起。

综合工具可以给出基于 FPGA 的专用乘法器的实现。例如，如果采用 Xilinx Spartan 器件，并用 Xilinx ISE 工具综合图 6.22 的 VHDL 代码，则会用 FPGA 的 4 个 18×18 乘法器。除此之外，还用到 FPGA 中的其他多个逻辑模块，这些逻辑模块用来实现部分积的求和运算。64 个 I/O 管脚既被用于提供被乘数和乘数，也用来进行输出。这里外部管脚用于提供乘法器的输入，但是乘法器的输入也可以来自 FPGA 中的内嵌存储器或其他可选寄存器。

```
library IEEE;
use IEEE.numeric_bit.all;

entity multiplier is
  port(A, B: in unsigned (31 downto 0);
       C: out unsigned (63 downto 0));
end multiplier;

architecture mult of multiplier is
begin
  C <= A * B;
end mult;
```

图 6.22　专用乘法器的 VHDL 代码

6.8　可编程能力的代价

FPGA 的可编程能力是以大量的硬件资源为代价的。在基于 SRAM 的 FPGA 中（例如 Xilinx 的 XC400 家族、Virtex 家族和 Spartan 家族 FPGA），SRAM 用于实现逻辑模块、可编程互连和可编程 I/O 模块。许多现代 FPGA 的逻辑模块都包含 4 变量函数生成器。一个 4 变量函数生成器需要 16 比特 SRAM。逻辑函数可以通过 LUT 中填入适当的比特来实现。此外，还有几个多路选择器用于选择这些生成函数的输出，或者用于控制锁存输出还是直接输出，或者用于生成更多变量的函数。一个 2 选 1 多路选择器需要 1 比特 SRAM 用于选择输入，4 选 1 多路选择器需要 2 比特 SRAM 用于选择输入。参阅图 6.23 的逻辑模块，带有 M 标志的小方块代表编程多路选择器的存储单元。一个存储单元用来选择外部时钟使能信号，另一个存储单元用来对时钟取反。该可配置逻辑模块总共需要 46 个存储单元。3 个函数生成器（LUT）中的 40 个存储单元可以用来实现一个简单的单变量函数，也可以用来实现一个复杂的 5 变量函数。

下面我们再用一个例子说明可编程性的开销。一个 Actel Fusion FPGA 的逻辑模块如图 6.15 所示。图中每个开关都需要一个 Flash 存储单元。我们需要这些 Flash 存储单元对逻辑模块进行编程，进而构成了逻辑模块的可编程能力。

I/O 模块也包含多个可编程点。参阅图 6.24 的 I/O 模块。控制配置的存储比特用带有 M 记号的方块表示。这些存储比特可以控制是否使用三态输出，是否对输出取反，是否使输出锁存，控制信号的转换速率，是否使用上拉电阻，等等。

每个 SRAM 单元一般要用 6 个晶体管。一个 Flash 存储单元的面积大约为一个 SRAM 面积的 25%。可编程点为 FPGA 增添了灵活性，但这一灵活性是以占用 SRAM 和 Flash 存储单元为代价的。表 6.3

列出了 Xilinx Spartan 和 Virtex FPGA 中配置比特的数量。XC2V40 是 Virtex II 族 FPGA，它包含 512 个 4 变量 LUT，需要 338 976 个配置比特。Virtex II 族的另外一种 FPGA（XC2V8000）含有 93 184 个 4 变量 LUT，需要 2600 万个配置比特。所以显而易见，FPGA 的灵活性和可编程性的代价是很高的。

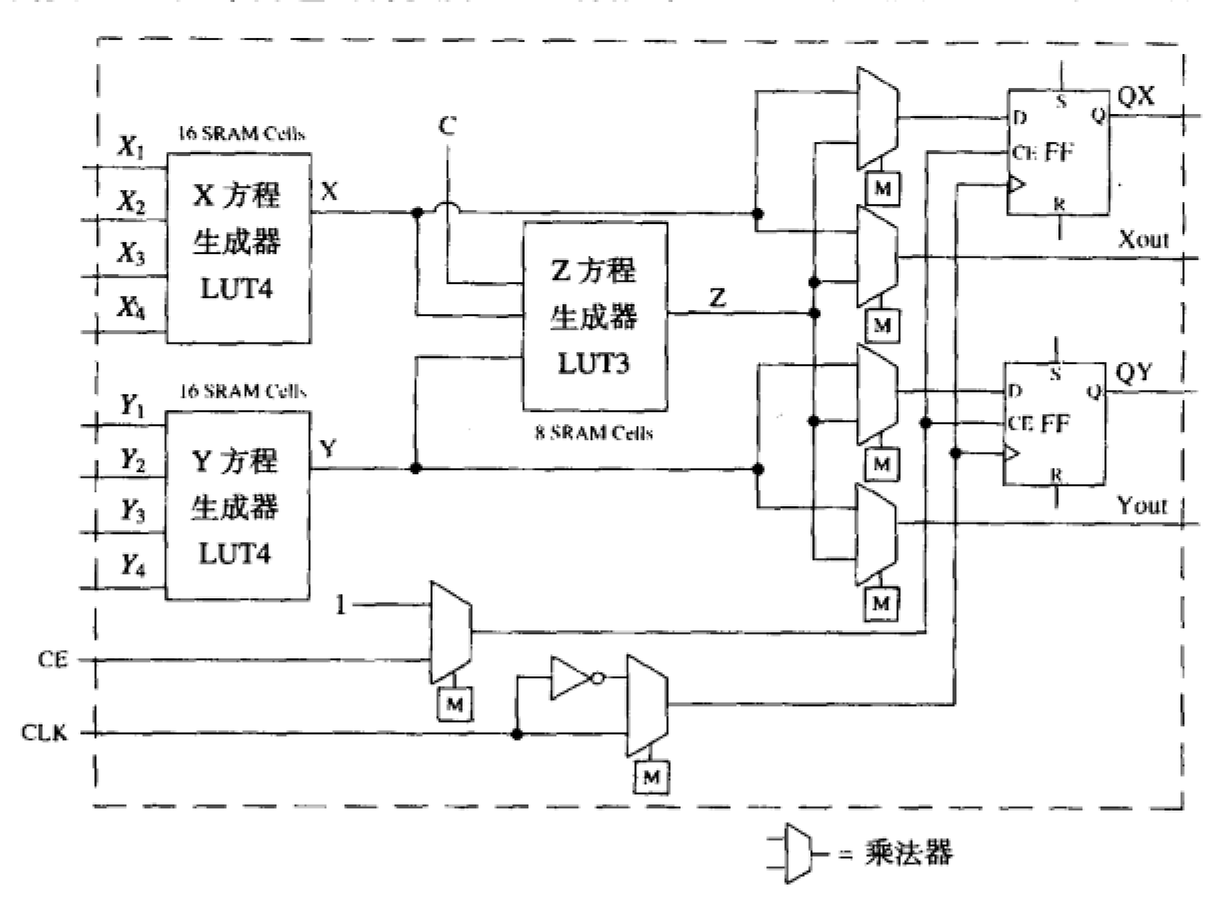


图 6.23 具有多个可编程 SRAM 单元的逻辑模块

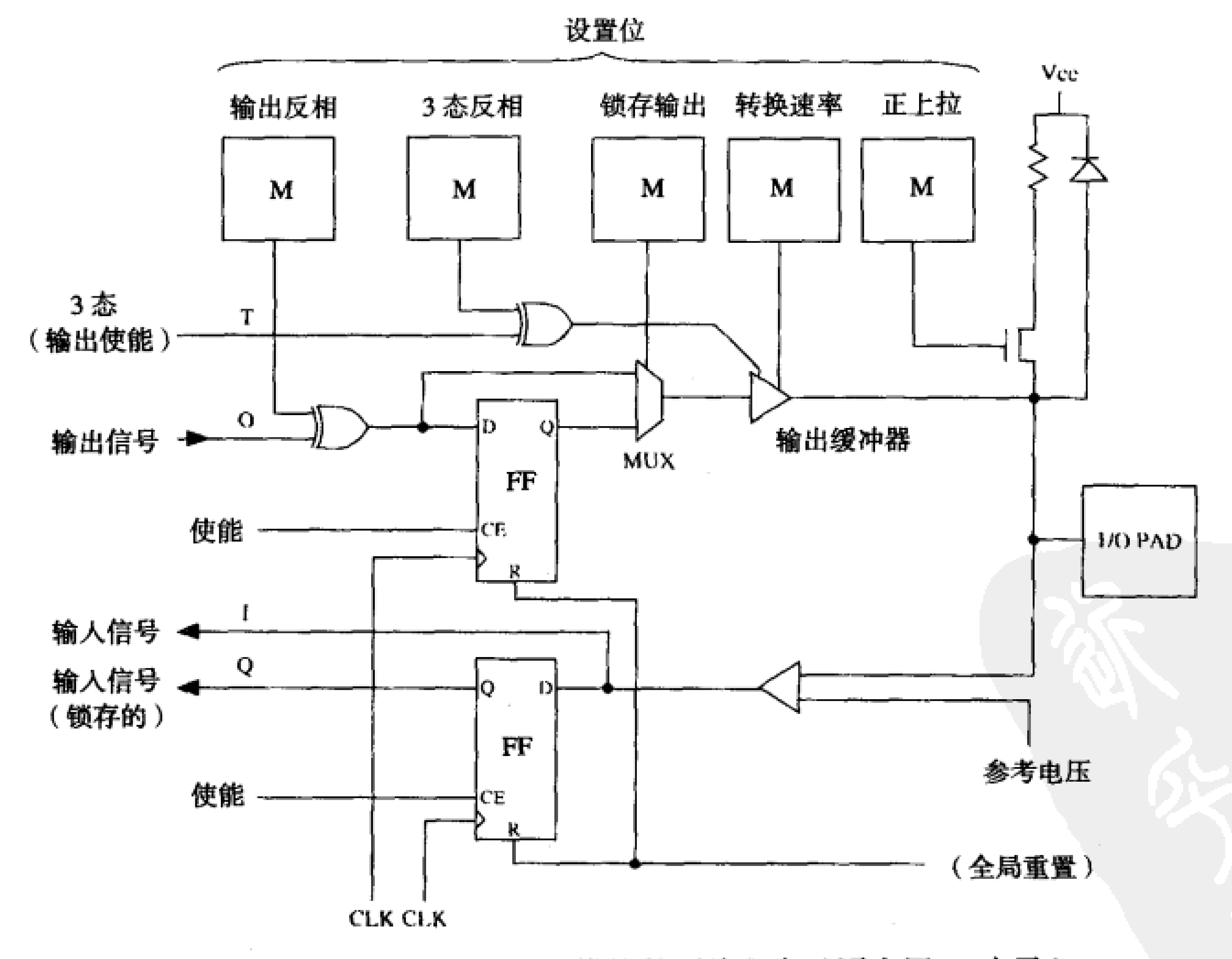


图 6.24 FPGA I/O 模块的可编程点（图中用 M 表示）

表 6.3 各种 FPGA 中配置比特数

生产厂商	器件家族	器件	配置比特数	逻辑模块个数	LUT 个数	可用 I/O 管脚个数
Xilinx	Virtex-5	XC5VLX30	8.4M	4800	19 200	400
		XC5VLX330	79.7M	51 840	207 360	1200
Xilinx	Virtex-II	XC2V40	0.3M	256	512	88
		XC2V8000	26.2M	46 592	93 184	1108
Xilinx	Spartan3E	XC3S100E	0.6M	960	1920	108
		XC3S1600E	6.0M	14 752	29 504	376
Altera	Stratix II	EP2S15	4.7M	6240	12 480	366
		EP2S180	49.8M	71 760	143 520	1170
Altera	Stratix	EP1S10	3.5M	10 570	10 570	426
		EP1S80	23.8M	79 040	79 040	1238
Altera	Cyclone II	EP2C5	1.3M	4608	4608	158
		EP2C70	14.3M	68 416	68 416	622

6.9 FPGA 和单热状态赋值

当使用 FPGA 进行设计时，减少所用到的触发器的数量可能并不是很重要，但是我们应该努力减少所用逻辑单元的总数，也应该努力减少各单元之间的互连数目。为了设计更快的逻辑电路，我们应该努力减少实现函数的单元数量。单热状态赋值法可以帮助解决这一问题。单热状态赋值法比编码赋值法使用更多的触发器，但是下一状态方程却更简单。

单热状态赋值的每个状态都使用一个触发器，因此一个 N 状态的状态机需要 N 个触发器。在每一个状态只有一个触发器置为 1 。例如，一个 4 状态(T_3, T_2, T_1 和 T_0)系统使用四个触发器(Q_3, Q_2, Q_1 和 Q_0)，其状态赋值如下：

$$T_0 : Q_0Q_1Q_2Q_3 = 1000, \quad T_1 : 0100, \quad T_2 : 0010, \quad T_3 : 0001 \tag{6.6}$$

其他 12 种组合不使用。

我们通过分析状态图或跟踪 S M 图的路径，就可以写出下一个状态和输出表达式。考虑图 6.25 给出的部分状态图。触发器 Q_3 的下一个状态表达式可以写为

$$Q_3^+ = X_1Q_0Q_1'Q_2'Q_3' + X_2Q_0'Q_1Q_2'Q_3' + X_3Q_0'Q_1'Q_2Q_3' + X_4Q_0'Q_1'Q_2'Q_3$$

然而，因为 Q_0=1 暗示着 Q_3=Q_2=Q_1=0，所以 $Q_1'Q_2'Q_3'$ 项就多余可以去掉。同理，所有取反的状态变量都可以去掉，这样下一个状态表达式简化为

$$Q_3^+ = X_1Q_0 + X_2Q_1 + X_3Q_2 + X_4Q_3$$

注意到每一项只包含有一个状态变量。同理，每个输出方程的每一项也只包含有一个状态变量：

$$Z_1 = X_1Q_0 + X_3Q_2 , \qquad Z_2 = X_2Q_1 + X_4Q_3$$

当使用单热状态赋值时，与每个指向下一个状态的弧线（或每一连接路径）对应，每个触发器的下一状态方程只含有一项。一般，每个下一状态方程和每个输出方程的每一项都只含有一个状态变量。

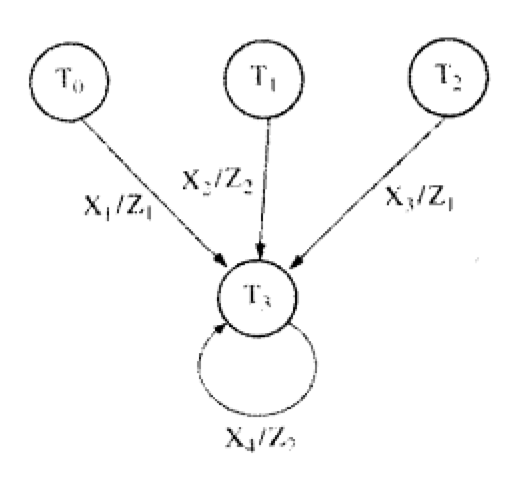


图 6.25　状态图

当使用单热状态赋值时，若要系统复位只需要将一个触发器置为 1，无需将所有触发器复位为 0。如果所用的触发器均无预先设置输入（如 Xilinx3000 系列），那么我们可以修改单热状态赋值，所有的 Q_0 用 Q_0' 来取代。对于之前的赋值，我们可以做如下修正：

$$T_0: Q_0Q_1Q_2Q_3 = 0000, \quad T_1: 1100, \quad T_2: 1010, \quad T_3: 1001 \tag{6.7}$$

修正后的表达式为

$$Q_3^+ = X_1Q_0' + X_2Q_1 + X_3Q_2 + X_4Q_3$$

$$Z_1 = X_1Q_0' + X_3Q_2, \qquad Z_2 = X_2Q_1 + X_4Q_3$$

如果不修改单热状态赋值，而且还要解决复位问题，那么我们可以使用另一个方法：在触发器方程中添加一项，使其在初始状态时为 1。给电后，若系统复位为状态 0000，则我们把 $Q_0'Q_1'Q_2'Q_3'$ 加到 Q_0^+ 方程中。这样，第一个时钟过后，状态将从 0000 变为 1000 (T0)，这是正确的初始状态。一般，我们同时试一试最少状态变量赋值和单热状态赋值，以找出使用逻辑单元数最少的赋值方法。如果运行的速度是主要因素，那么应该选择使逻辑单元速度最快的设计。当使用单热状态赋值时，需要较多的下一状态方程，但是通常下一状态方程和输出方程都包含较少的变量。一个含有较少变量的方程通常需要较少的逻辑单元来实现。级联的单元越多，传输延迟越长，操作速度就越慢。

6.10　FPGA 的容量：门的最大个数和门的可用个数

你经常会碰到 FPGA 门的个数问题。你也知道，很多 FPGA 并不是门阵列结构的，有些只是 LUT 阵列结构，而不是门阵列结构。因此 FPGA 的门个数到底是指什么？

对 FPGA 使用者来说，并不感兴趣构建 FPGA 的最基本门的个数，它也不是一个有用的度量。使用者所关心的是 FPGA 含有的专用电路的个数，我们称之为等效门个数。但是，你也知道，这种电路个数计算，与电路的类型、电路元件之间互连的类型等因素有关。

门个数的估算有很多不同方法。等效门个数的一个近似估算可以以逻辑模块为单位，先考虑一个逻辑模块上能实现的电路，然后再根据逻辑模块个数估算 FPGA 总的门个数。用这种方法估算得到的门个数比 FPGA 中实际实现的电路的门个数要多。一个更好的估算方法是基于基准电路的估算。PREP 是一个为 ASIC 和 FPGA 的规范而制定标准基准电路的一个组织。假设在 ASIC 上实现一个特定的电路需要 2000 个门，且一个 FPGA 中可以包含 20 个这样的电路，那么 FPGA 生产商可以估计出该 FPGA 的最大门个数为 40 K。由于我们只简单重复计算这种电路的个数，而并

没有考虑电路之间的实际互连，所以用这种方法数出的门个数比 FPGA 中实际实现电路的门个数要多。所以一些 FPGA 除了提供最大门个数外，还有一些加权算法。PREP 收集和推广的基准电路对 FPGA 的规范是很有帮助的。

因为 LUT 可以实现不同的逻辑电路，我们很难估算出基于 LUT 的 FPGA 中门的个数。一个 4 输入 LUT 可以用来实现一个 4 变量逻辑表达式，该逻辑表达式可以由一个或多个乘积项构成。一个 4 输入 LUT 还可以用来存储 16 比特的信息。当 LUT 作为 RAM 使用时，所估计的门个数可能会比实际的高。因此根据 LUT 中作为 RAM 使用的部分的多少，我们对同一个 FPGA 可能会估计出不同数目的门个数。生产商通常把 CLB 中的一部分（20%～30%）看做是 RAM，然后再计算该 FPGA 的“系统门”个数。

Altera 为它们生产的 APEXII 家族提供了两种门个数：门的最大个数和门的可用个数。APEXII 中所含门的最大个数为 190 万～525 万，但是通常认为其门的个数为 600 千～300 万个。有些 FPGA 生产商把逻辑模块数量作为芯片容量提供给客户，而不计算门的数量。

PREP 基准电路

PREP（Programmable Electronics Performance Company）是一个非盈利性组织，他们为可编程 ASIC 收集和推广一系列基准电路。PREP 1.3 的 9 个基准电路为

1. 由 1 个 4 选 1 MUX、1 个寄存器和 1 个移位寄存器组成的 8 位数据通道。
2. 由 2 个寄存器、1 个 4 选 1 MUX、1 个计数器和 1 个比较器构成的 8 位时间计数器。
3. 1 个小状态机（8 状态、8 输入和 8 输出）。
4. 1 个大状态机（16 状态、8 输入和 8 输出）。
5. 由 1 个 4×4 乘法器、1 个 8 位加法器和 1 个 8 位寄存器构成的 ALU。
6. 1 个 16 位累加器。
7. 1 个具有同步置入和使能端的 16 位计数器。
8. 1 个具有置入和使能端的 16 位可预置算计数器。
9. 1 个 16 位地址译码器。

PREP 的在线信息中包括 Verilog 和 VHDL 的源代码和测试平台（由 Synplicity 提供）。PREP 还提供综合基准电路，包括位片处理器、乘法器和 R4000 MIPS RISC 微处理器。

6.11　设计综合

在前几节中，我们把一些设计手工映射到 FPGA 逻辑模块中。这一过程类似于编写微处理器的汇编语言程序，是很繁琐的。若设计者只能通过如此方式进行设计，则它们的完成效率将会非常低。现在我们可以用高级语言编程（如 C 语言），再用编译器进行编译转换。同样，现在数字系统也可以在行为或 RTL 层面上进行设计，然后再转换到目标器件上去。这种技术不仅在 FPGA 上得到应用，而且在 ASIC 设计上也得到应用。

现在很多的 CAD 工具可以根据一个数字系统电路的 VHDL 描述，自动生成实现该系统的电路。综合是指把抽象的上层设计转换为底层电路描述（通常为逻辑电路图）。CAD 工具的输入为行为描述或结构描述的 VHDL/Verilog 模块。综合工具的输出可以是逻辑电路图和相关的连线表，它是用门电路、触发器、寄存器、计数器、多路选择器、加法器和其他基本逻辑模块间的互连关系来表示这一数字系统的实现。这种表示一般称为网表（netlist）。这一电路都可以在 FPGA, CPLD

和 ASIC 上实现。

一般计算机辅助设计流程为以下几步：

设计转换（综合）和优化。
映射。
布局。
布线。

这些步骤示于图 6.26。在本节中，我们介绍设计转换和优化技术。设计的映射、布局和布线将在以下节中介绍。

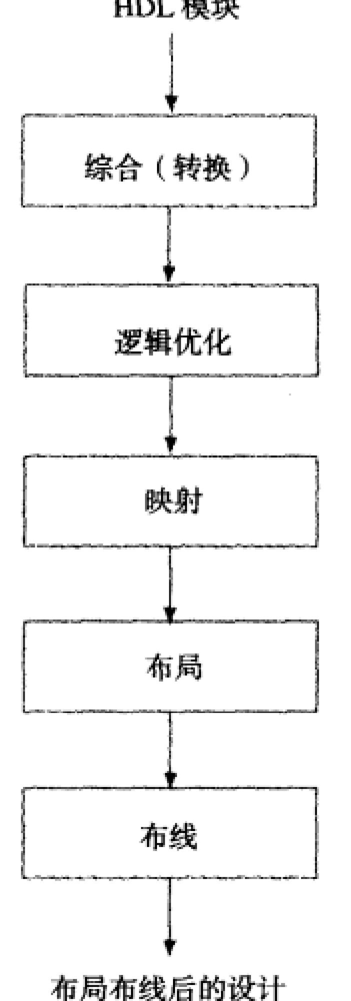


图 6.26 CAD 设计流程

一个 VHDL 代码即使能通过编译和仿真，但也不一定能正确综合。甚至即使它可以正确综合，也不一定就很有效的实现。通常，综合工具只支持部分 VHDL 代码，其他部分还要做修改以使综合工具“明白”设计者的意图。为了有效率地实现系统，还要对 VHDL 代码进行进一步的优化修正。

在 VHLD 描述中，一个信号可以表示触发器或寄存器的输出，也可以表示组合逻辑模块的输出。综合工具通过上下语句确定该信号所代表的含义。如下面的并发语句：

```
A<= B and C;
```

暗示 A 应该使用组合逻辑实现。另一方面，若下面的顺序语句：

```
wait until clock'event and clock = '1';
A<= B and C;
```

出现在进程语句中，则它暗示 A 代表一个寄存器（或触发器），在时钟上升沿改变状态。

当使用整数信号时，设定整数的范围是很重要的。如果没有设定范围，由于 VHDL 中整数的最大范围是 32 位，所以 VHDL 综合器就可能把该整数信号当做一个 32 位寄存器。给定整数范围后，大多数综合器都使用适当位数的二进制加法器实现整数的加法和减法运算。

大多数 VHDL 综合器对 VHDL 代码先逐行转换为门电路、寄存器、多路选择器和其他基本单元，几乎不做任何优化。转换之后，再对电路进行优化。综合器把特定的 VHDL 结构和特定的硬件结构联系在一起。例如 **case** 语句基本上转换为多路选择器，“+”、“-”和比较操作转换为加法器，移位操作转换为移位寄存器，等等。

在 VHDL 代码进行转换和优化的初期，综合工具会从可用库中选择元件。为了可以通过不同技术实现系统，应该提供不同的元件库。

6.11.1 case 语句的综合

图 6.27 给出了 Synopsis 的设计编译器（Design Compiler）把一个 **case** 语句用多路选择器和门电路实现的一个例子，程序代码示于图 6.27(a)。整数 a 和 b 分别用一个 2 位二进制数来表示。需要使用两个 4 选 1 多路选择器。a 的两位用做多路选择器的控制输入。多路选择器的输入端与逻辑 1 和逻辑 0 硬件相连。6.27(b)给出了典型综合器生成的硬件电路。

大多数现代综合器都进行逻辑化简优化。由于多路选择器输入为常数，通过检查图 6.27(c)的真值表，可以把多路先择期和几个门去掉。优化后的输出方程为 $b_1 = a_1'a_0 = (a_1 + a_0')'$ 和 $b_0 = (a_1a_0')'$。图 6.27(a)代码优化后的电路，由一个 NOR 门、一个 NAND 门和一个 NOT 门构成，示于图 6.27(d)。

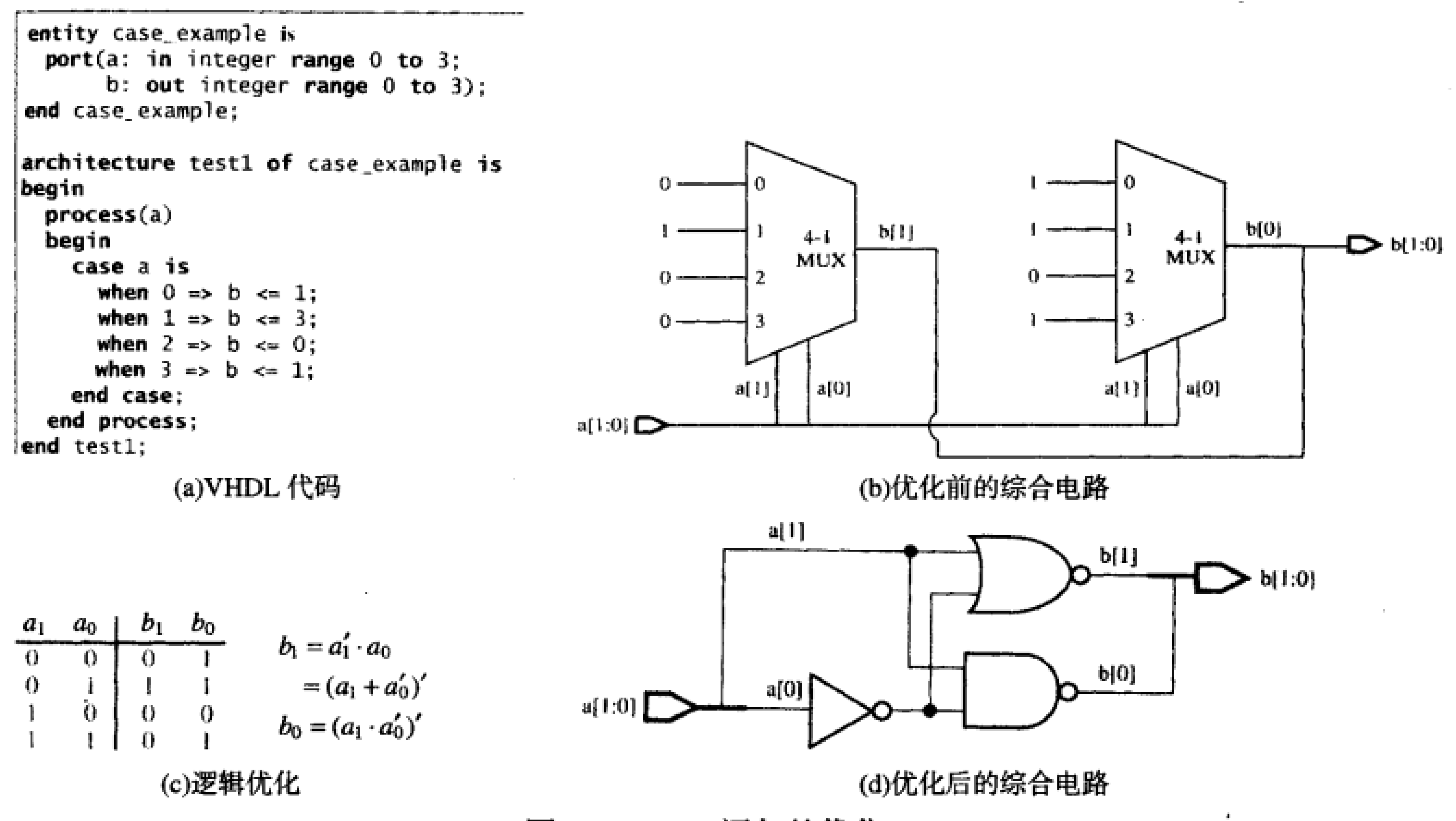


图 6.27　case 语句的优化

锁存器的无意生成

通常，当一个 VHDL 信号被赋值后，它会一直保持该值，直至再次被赋值。由于这个性质，一些 VHDL 综合器一般是生成锁存器，这时设计者并不想锁存该信号。图 6.28(a)给出了由 case 语句生成一个不需要的锁存器的例子。Case 语句应生成一个 4 选 1 多路选择器，其数据输入在 case 语句的不同分支中给出。控制线由 a 的值控制。由于当 a 不等于 0, 1 或 2 时 b 的值没有设定，所以综合器就认为当 a=3 时，b 的值应该用锁存器保存起来。

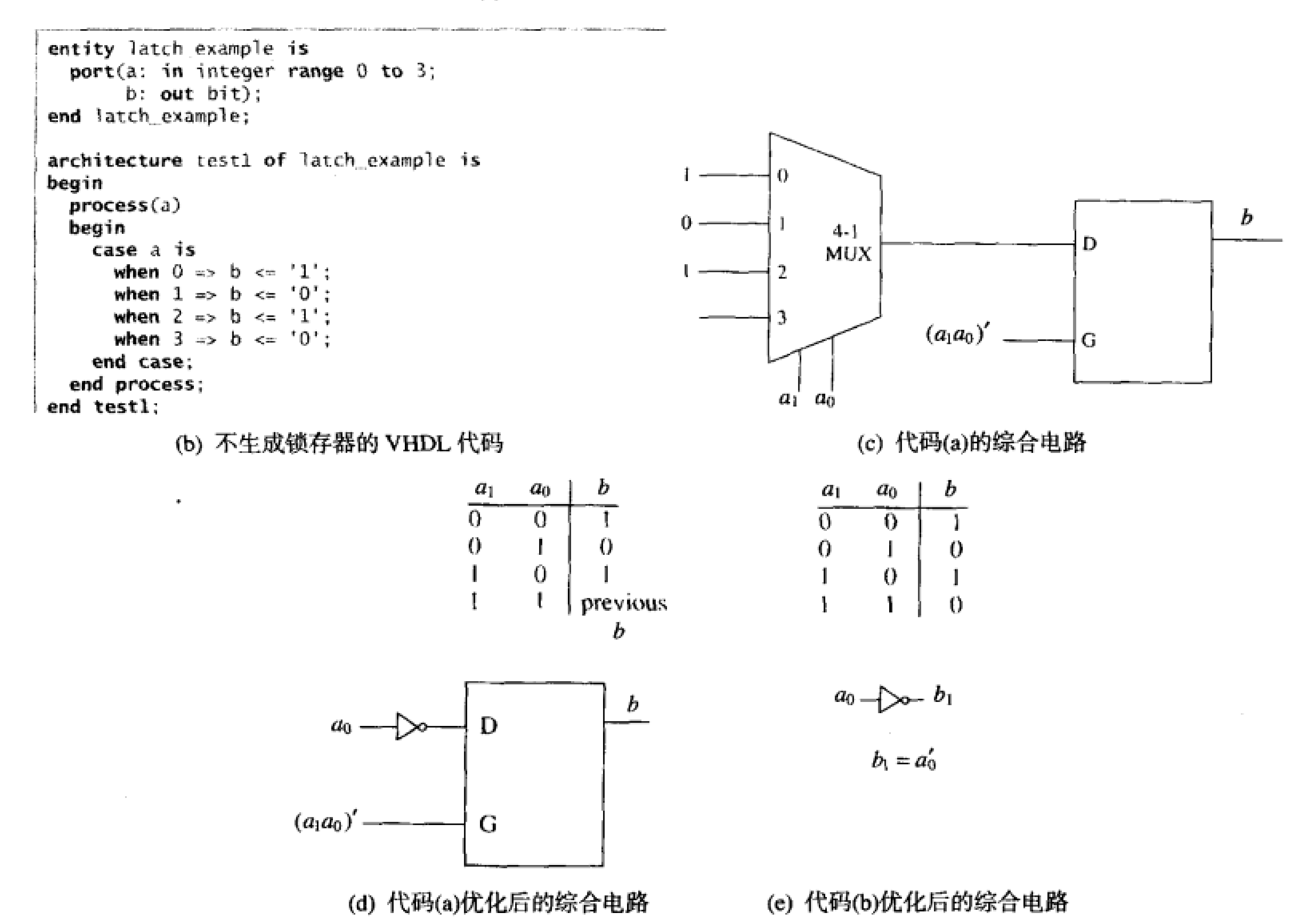
当 a=3 时，前一个 b 的值就作为输出。这就迫使必须要有一个锁存器，其输入 $D= a_0$。为了在锁存器中存储数据，所以当 a=3 时，锁存器控制信号 G 应该为 0。这样 $G=(a_1a_0)'$。一个简单的综合器可能会生成一个 4 选 1 多路选择器和一个锁存器，如图 6.28(c)所示。通过在 VHDL 代码中用语句 b<='0'取代 null（如图 6.28(b)所示），我们就可以把锁存器去掉了。如果对代码做如上修改，那么大多数综合器就只会生成一个多路选择器，而不会生成锁存器。

大多数现代综合器都进行了逻辑化简优化。例如，该电路并不需要 4 选 1 多路选择器。为了优化该电路，可以分析图 6.28(d)的真值表。我们马上看到：当 a 等于 0, 1 或 2 时，$b = a_0'$。对于该代码，一个具有优化功能的综合器可能只生成一个 NOT 门，如图 6.28(e)所示。如果代码中的 null 语句没有去掉，那么该综合器也生成锁存器，如图 6.28(d)所示。

```
entity latch_example is
  port(a: in integer range 0 to 3;
       b: out bit);
end latch_example;

architecture test1 of latch_example is
begin
  process(a)
  begin
    case a is
      when 0 => b <= '1';
      when 1 => b <= '0';
      when 2 => b <= '1';
      when others => null;
    end case;
  end process;
end test1;
```

(a) 生成锁存器的 VHDL 代码




```
entity latch_example is
  port(a: in integer range 0 to 3;
       b: out bit);
end latch_example;

architecture test1 of latch_example is
begin
  process(a)
  begin
    case a is
      when 0 => b <= '1';
      when 1 => b <= '0';
      when 2 => b <= '1';
      when 3 => b <= '0';
    end case;
  end process;
end test1;
```

(b) 不生成锁存器的 VHDL 代码　　(c) 代码(a)的综合电路

a_1	a_0	b
0	0	1
0	1	0
1	0	1
1	1	previous b

a_1	a_0	b
0	0	1
0	1	0
1	0	1
1	1	0

(d) 代码(a)优化后的综合电路　　(e) 代码(b)优化后的综合电路

图 6.28　锁存器的无意生成例子

6.11.2　if 语句的综合

使用 **if** 语句时，我们应该注意给定每个分支的信号赋值。比如，当设计者写下以下语句时，

```
if A = '1' then Nextstate <= 3; Z<= 1;
end if;
```

他或她可能想 $A \neq 1$ 时让 Nextstate 保持前面的值，该代码也正确仿真。然而，综合器可能把该代码理解为：当 $A \neq 1$ 时，Nextstate 未知（'X'），这样综合结果就可能出错。综合器还会为 Z 生成一个锁存器。由于这个原因，我们最好在使用每个 if 语句时都要包含 else 语句。例如，

```
if A = '1' then Nextstate <= 3; Z<=1;
  else Nextstate <= 2; Z<= 0;
end if;
```

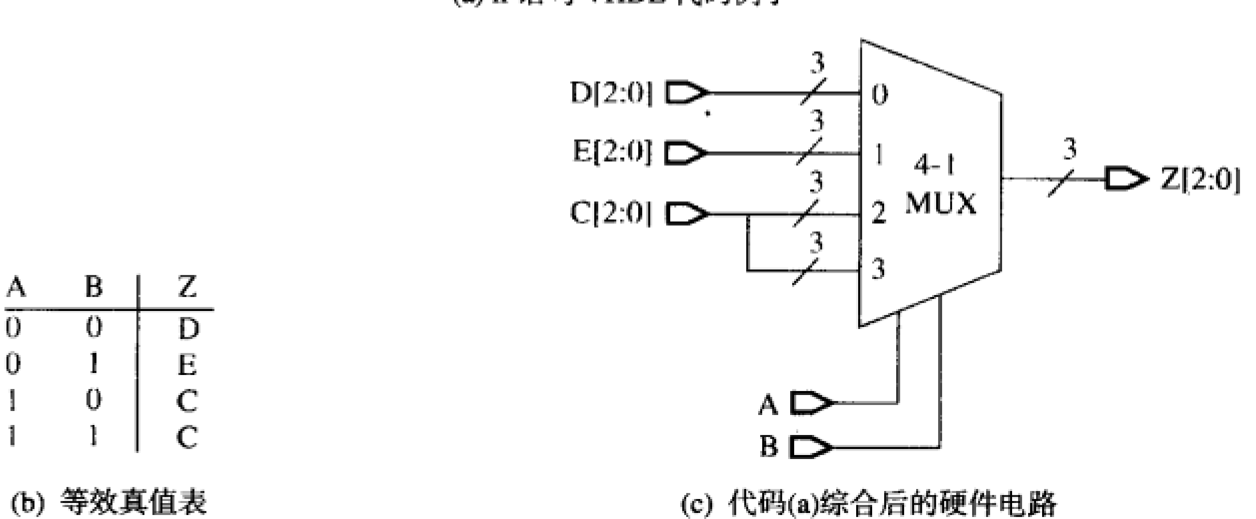
这时就不会含糊不清了。

图 6.29 给出了一个典型的综合器如何把一个 **if-then-elsif-else** 语句用多路选择器和门电路实现的例子。各种输入组合对应的真值表见图 6.29(b)。当 *A*=1 时，*C* 被选中；当 *A*=0 且 *B*=0 时，*D* 被选中；当 *A*=0 且 *B*=1 时，*E* 被选中。综合后的硬件电路示于图 6.29(c)。*A* 和 *B* 用做多路选择器的选择信号。

```
entity if_example is
  port(A, B: in bit;
       C, D, E: in bit_vector(2 downto 0);
       Z: out bit_vector(2 downto 0));
end if_example;

architecture test1 of if_example is
begin
  process(A, B)
  begin
    if A = '1' then Z <= C;
    elsif B = '0' then Z <= D;
    else Z <= E;
    end if;
  end process;
end test1;
```

(a) if 语句 VHDL 代码例子



A	B	Z
0	0	D
0	1	E
1	0	C
1	1	C

(b) 等效真值表

(c) 代码(a)综合后的硬件电路

图 6.29 if 语句的综合

例 LE<=(A<=B) 语句生成的硬件电路是什么？假设 *A* 和 *B* 均为 4 位矢量。

解：这是一个 4 位比较器。两个<=符号中只有一个是用做赋值的。*A* 和 *B* 之间的'<='是关系运算符。当 *A* 小于或等于 *B* 时，赋值符的右边返回 TRUE 或'1'。因此，当 *A* 小于或等于 *B* 时，LE 置为 1，否则 LE 置为 0。

大多数标准比较器输出为 EQUAL_TO(EQ), GREATER_THAN(GT)和 LESS_THAN(LT)。在本例中，当 EQUAL_TO 或 LESS_THAN 为真时，LE 为'1'。硬件结构示于图 6.30。

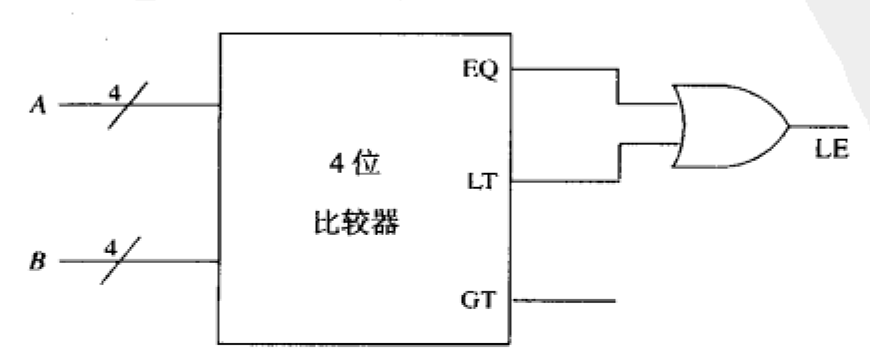


图 6.30 小于等于检测器的硬件电路

6.11.3 算术单元的综合

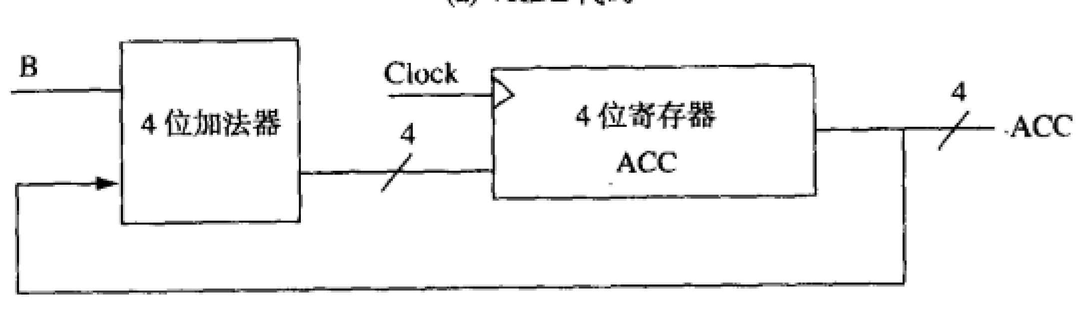
CAD 工具为综合提供很多设计库，包括实现定义在 numeric 包集合中的各个操作的元件。图 6.31 所示例子使用 IEEE numeric_std 库。当图中代码综合时，生成的元件包括 4 位比较器、带有 4 位累加器的 4 位二进制加法器和 4 位计数器。有些综合工具会将计数器用一个输入为“0001”的 4 位加法器来实现，然后对其进行优化去掉多余的门。最终的硬件电路如图 6.31(b)所示。

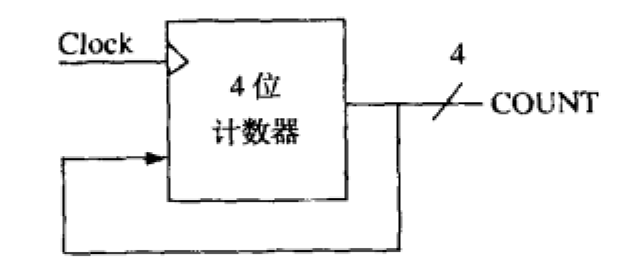
```
library IEEE;
use IEEE.numeric_bit.all;

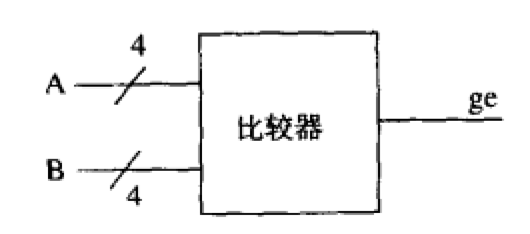
entity examples is
  port(signal clock: in bit;
       signal A, B: in signed(3 downto 0);
       signal ge: out boolean;
       signal acc: inout signed(3 downto 0) := "0000";
       signal count: inout unsigned(3 downto 0) := "0000");
end examples;

architecture x1 of examples is
begin
  ge <= (A >= B);   -- 4-bit comparator
  process
  begin
    wait until clock'event and clock = '1';
    acc <= acc + B;   -- 4-bit register and 4-bit adder
    count <= count + 1;   -- 4-bit counter
  end process;
end x1;
```

(a) VHDL 代码







(b) VHDL 代码综合后的硬件电路

图 6.31 VHDL 代码综合示例

例 优化生成语句 EQ3<= (A=3)的硬件电路，假设 A 为 4 位矢量。

解： 该语句可以用一个 4 位比较器来实现。比较器的一个输入为 A，另一个输入为数字 3（0011）。

然而，由于比较器的一个输入为常数 3，所以我们可以把它优化为一个 AND 门和两个反相器，如图 6.32 所示。

有些综合器不能自动优化硬件。在这种情况下，可以把 VHDL 源代码改为

```
EQ3 <= not  A(3)  and not  A(2)  and  A(1)  and
A(0);
```

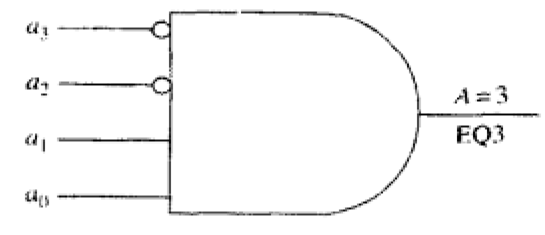


图 6.32　等效检测器优化后的硬件

该语句综合后的硬件结构就会是图 6.32 所示电路。

对于不同的目标技术需要使用不同的优化方法。例如，对于基于门电路的 FPGA 来说，降低门的绝对数量是很重要的；但是如果对于基于 LUT 的 FPGA 来说，优化时完全没必要考虑门的绝对数量，只需考虑 LUT 的数量就可以了。

6.11.4　面积、功耗和延迟的优化

大多数 VHDL 综合器允许对设计的最大速度或最小逻辑片（slice）面积进行优化。与面积和延迟一样，现在功耗也逐步成为一个主要的设计约束。一般来讲，一个参数的优化带来另一个参数的劣化。比如，如果我们改进了速度，那么可能会增大面积。改进速度意味着一些原本是重复使用一些门执行的串行操作必须要并行执行了。因此，一般速度的改进会带来元件数的增加。例如，一个并行 4 位加法器组合电路要比一个 4 位串行加法器使用更多的硬件，以实现较快的速度。如果我们对面积进行优化，那么就必须减少元件的数量，由此反而增加了关键路径的长度。关键路径是指电路中的最长延迟路径。

CAD 工具提供逻辑门库。这些库提供了各种选择以满足在面积、速度和功耗方面的要求。这些逻辑门和基本模块，针对面积、速度和功率分别进行了优化，或者针对其中的两项或多项进行了优化。设计者根据指标要求，可以选用库中的合适元件。

电路的面积和延迟通常是负相关关系。能量和延迟也是负相关关系。我们常用面积-时间（AT）积和能量-延迟（ED）积来度量电路的性能。AT^2 和 ED^2 也可以作为度量电路和系统性能的标准。

尽管面积和延迟或者能量和延迟有负相关关系，但我们也可以对面积、延迟和功耗同时进行优化。比如，参阅图 6.27(b)～(d)和图 6.28(c)～(e)的优化过程。这些优化是在逻辑层上进行的，以更有效的方式实现了设计要求，使用了更少的硬件、占用更少的面积、消耗更少的功耗和惊奇的短关键路径。

当使用 FPGA 进行设计时，我们必须要时刻牢记对个别门的优化并不是 FPGA 的最佳优化。在一个 SRAM FPGA 中，最重要的是表达式的变量个数要最少，而不是乘积项的个数最少，因为在 LUT 中已经存储了该表达式的全部真值表。

CAD 工具主要生产商	FPGA CAD 工具主要生产商
Cadence	Xilinx
Synopsis	Altera
Mentor Graphics	Actel

6.12 映射、布局和布线

设计一旦综合转换并生成网表后，就必须映射到特定实现技术中去。这一实现技术包括门阵列、FPGA、CPLD 和 ASIC 等标准单元设计。映射、布局和布线是把一个设计从网表转换到适当技术的三个主要步骤。

6.12.1 映射

映射是把一个与具体实现技术无关的电路和与具体某一目标实现技术有关的电路进行绑定。众所周知，一个设计可以用多种方法来实现：使用多路选择器、使用 ROM 或 LUT、使用 NAND 门、使用 NOR 门或者使用 AND-OR 门。设计也可以合用这几种技术来实现。

如果我们使用基于标准单元的门阵列，那么网表就需要映射到标准单元；如果使用基于 LUT 的 FPGA，那么设计就要映射到 LUT 上；如果我们使用只包含 4 选 1 多路选择器的 FPGA，那么设计就必须映射到一个只用多路选择器就可以实现的结构上来；如果目标技术只包含 2 输入 NAND 门，那么设计就必须映射到一个只用两输入 NAND 门就可以实现的结构上来。在本章的开始，我们通过使用一个移位寄存器和多路选择器，手工完成了这一映射过程。CAD 工具使用映射软件来完成这一任务。

标准单元法 标准单元设计是集成电路设计的常用技术。设计会映射到包含标准逻辑门的库中。通常标准逻辑门包括 NOT, AND, NAND, OR, NOR, XOR 和 XNOR 等。支持标准单元设计法的 CAD 工具通常也会包含一个库，库中放置复杂函数和标准构建模块（例如多路选择器、译码器、编码器、比较器和计数器等）。设计将映射到一个只使用库中单元就可以实现的结构上来。这些单元都按行排列，并由布线通道隔开，如图 6.33 所示。有些单元只可以用做行与行单元之间的布线。这些单元称为馈通单元。为了有效率地使用标准单元法，每个单元的高度必须一致。但是，这些单元也可以包括存储器模块、特殊算术模块等。

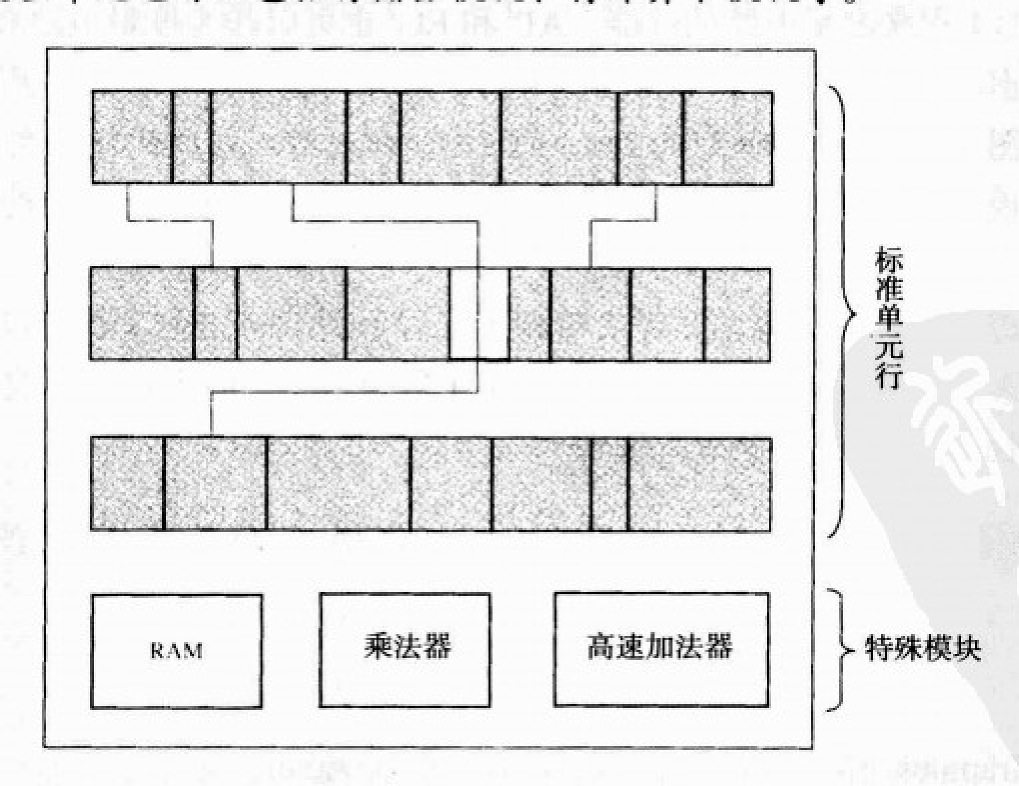


图 6.33 标准单元设计概况

6.12.2　布局和布线

布局是指把映射中确定的逻辑和 I/O 模块分配到目标实现硬件的各个位置上。它决定了每个子模块在硬件上的位置。这个过程很重要，因为它直接影响到下一步的布线。一个好的布局算法都要千方百计地减少设计的面积和延迟。设计的面积和延迟在一定程度上依赖于布线的好坏。算法通常可以估计布线的长度，并决定如何进行布局。复杂的布局算法最好不要采用，因为其占用了太多的运行时间。

布线是指把设计的各个子模块链接在一起的过程。布线很大程度上依赖于布局。因此布线和布局通常是共同进行的。布线可以通过多步进行。先进行全局布线以使布线线长度最短，然后再进行各个部分的具体布线。如果只有电路的一部分有变化，那么递增式走线的布线就很有用处了。

通常使用试探法进行布局，从最初的布局开始，通过反复试探其他的布局进行改进。比如，交换一个布局中的两个模块可以得到另外一种布局，再计算比较这两种布局的线长。这种试探过程一直进行到没有进一步的改进。

模拟退火技术也用于布局和布线中。退火最初是冶金学的一个名词。对大状态空间，模拟退火算法可以给出更快更好的优化解。模拟退火法并不能保证一定得到最佳的结果，但是它比穷举搜索更快接近全局最优解。模拟退火从一个可行的解开始（即合理的并不是最佳的），通过对布局进行随机改变（如置换）搜索更好的结果。迭代更新算法的每一步都只接受更好的结果。这种只接受更好结果的算法称为**贪婪（greedy）算法**。但是，如果我们只接受更好的结果，那么我们可能只能得到局部最优解。其实，有时候若接收“坏移动”，反而有利。通常这些“坏移动”会让我们最终达到全局最优解。

接受一个坏移动会带来冒险。在模拟退火进程开始阶段，我们还可以有更多的冒险，但在进程的后一阶段，我们就应该谨慎了，因为我们可能没有足够的时间把结果优化到可接受程度。就像在冶金学的物理退火中一样，在模拟退火算法中算法也有一个温度的概念。温度在开始时很高，随后不断降低。模拟退火算法允许存在依赖于温度的冒险。当温度下降时，接受坏点的能力也随之降低，最终算法变为贪婪算法，只能接受正确的结果。模拟退火算法与迭代更新算法的不同之处见图 6.34。*Y* 轴表示得到结果的消耗，*X* 轴表示进程进行的步数。

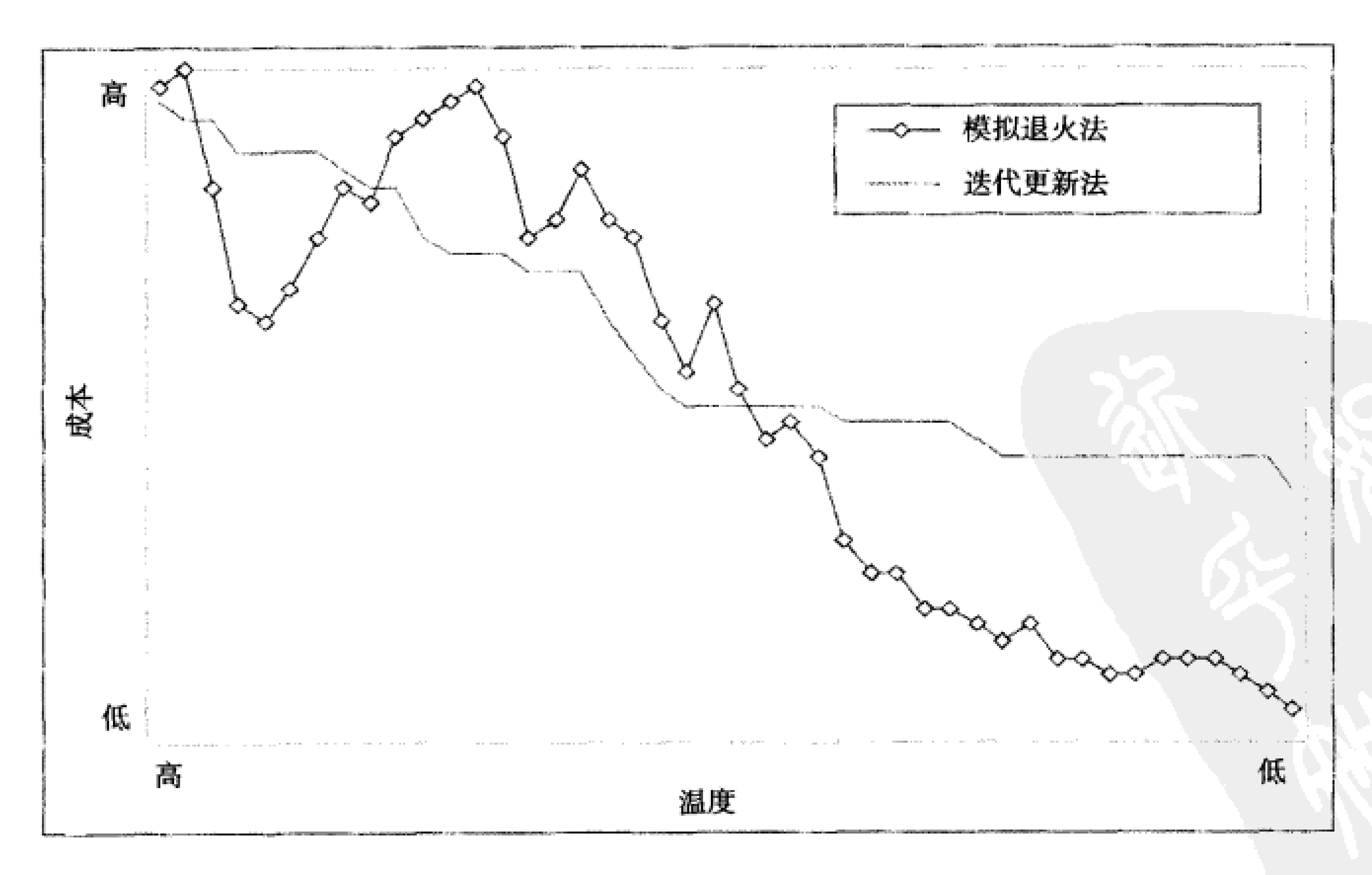


图 6.34　模拟退火法与迭代更新算法之比较

在模拟退火布局和布线算法中，先假设初始布局，再评估新布局。通常，对布局的评估是以所需走线的多少来进行的。如果通过一步移动可以得到更好的评估值（比如线长），那么我们认为这一步是较好的。

工具进行设计映射和布线的能力取决于所使用工具中的算法和资源的粒度。图 6.35 给出了一个使用 FPGA 进行数字系统设计的实例，该设计已经进行了布线（本例实际上是第 5 章的骰子游戏，使用早期的 Xilinx FPGA——XC3000 实现）。最上面的一个个小方块是 I/O 模块。从图中明显可以看出只有左上角和底部的一些 I/O 模块被使用了。中间的逻辑模块很多被使用了，但是仍有一些逻辑模块未被使用。综合工具将给出逻辑模块使用的数量和占全体模块的比例，并且给出触发器使用数量和占全体触发器的比例。

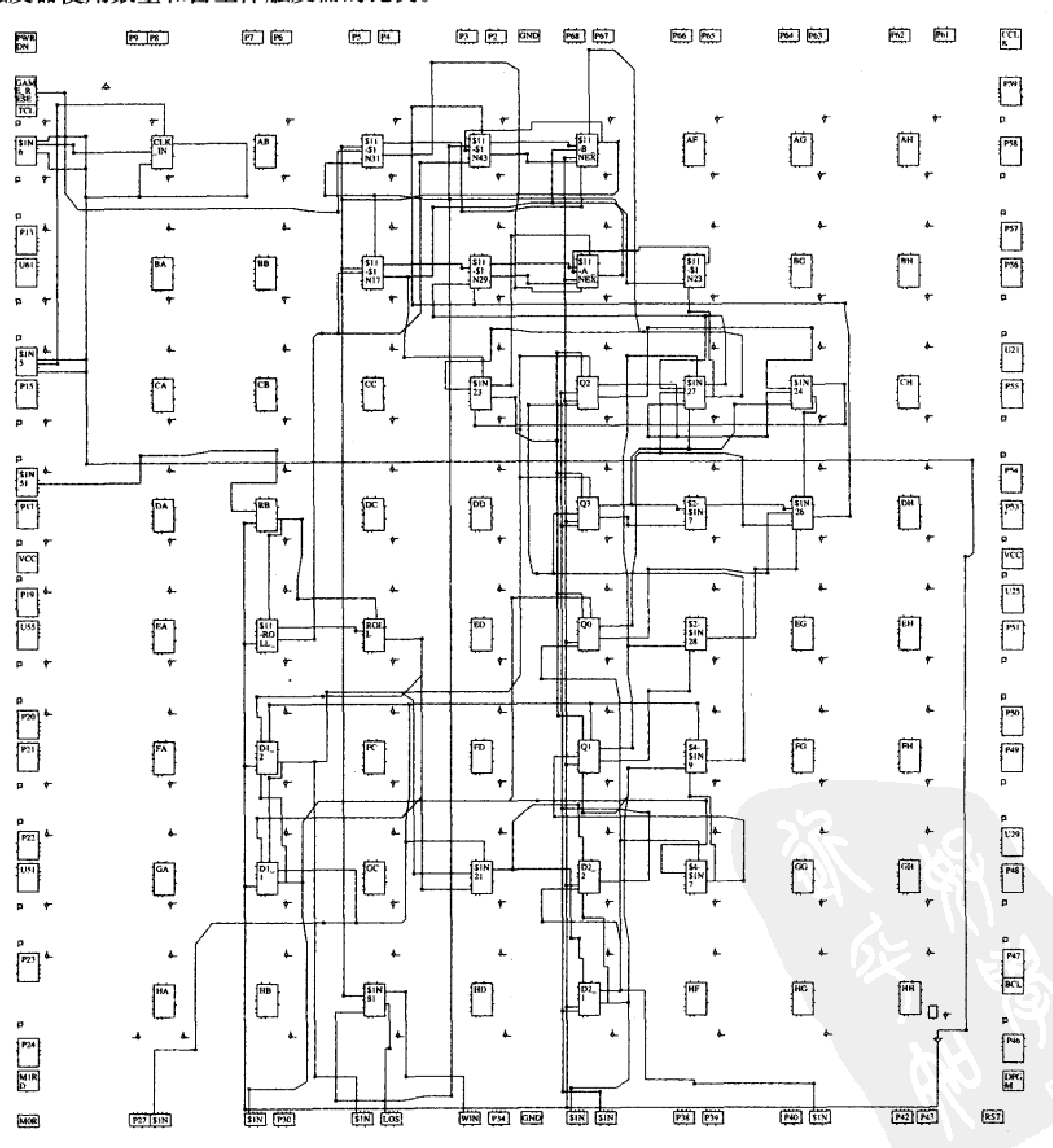


图 6.35　一个布线后的 FPGA

选择一个 FPGA 时，要考虑其逻辑模块的特点、映射工具的有效性、布线资源、布线工具的有效性等因素。如果逻辑模块的粒度很大，那么很可能有很大一部分逻辑模块将不会被使用。例如，我们看图 6.5 所示移位寄存器的例子，此例中很大一部分函数生成器就未被使用。再看图 6.2 和图 6.4 所示多路选择器的例子，逻辑模块中的触发器也未被使用。

本章中我们讨论了几种 FPGA 和用这些器件进行设计的过程。现在有很多高级 CAD 工具可以帮助我们用可编程门阵列进行设计系统。但是，本章中我们介绍了几个手工设计的实例，进而说明如何用 CAD 工具进行设计。同时，本章中还介绍了把多变量函数分解为几个具有较少变量函数的技术。现代 FPGA 的特性，例如内嵌存储器、内嵌多路选择器、进位链和级联链等，在本章中也加以介绍。此外，本章中还简单介绍了综合、映射、布局和布线操作。

习题

6.1 一个 8 位并行载入右移寄存器要求使用如图 6.1(a)所示 FPGA（带逻辑模块）实现。触发器记为 $X_7X_6X_5X_4X_3X_2X_1X_0$。控制信号 N 和 S 操作如下：$N=0$，不工作；$NS=11$，右移；$NS=10$，加载。右移的串行输入是 SI。

(a) 需要使用多少逻辑模块？

(b) 画出模块（图 6.1(a)中最右边的模块）所需的连接。连接 N 和 CE。

(c) 给出此模块的函数生成器输出。

6.2 使用图 6.1(a)所示逻辑单元实现一个 2 位的二进制计数器。$A0$ 是最低有效位，$A1$ 是最高有效位。计数器有一个同步加载(Ld)。计数器运行如下：

$En=0$	不改变。
$En=1$，$Ld=1$	在时钟的上升沿将外部输入 U 和 V 加载到 $A0$ 和 $A1$。
$En=1$，$Ld=0$	在时钟上升沿的计数器增 1。

(a) 给出 $A0$ 和 $A1$ 的下一状态方程。

(b) 给出所有图 6.1(a)的输入和连接，用粗线画出所需连接路径。使用输入 CE。并给出每个 4 输入 LUT 实现的函数。

6.3 一个 4 位右移寄存器要求使用如图 6.1(a)所示逻辑模块的 FPGA 实现。如果 $Ld=1$，$En=1$，则寄存器在时钟沿加载。当 $Ld=0$，$En=1$ 时，右移；当 $En=0$ 时，不改变。Si 和 So 是寄存器的移位输入和输出。$D_{3\text{-}0}$ 和 $Q_{3\text{-}0}$ 分别表示并行输入和输出。现最左边的触发器的下一状态等式是 $Q_3^+=En'Q_3+En(Ld\,D_3+Ld'\,Si)$。

(a) 给出其他三个触发器的下一状态方程。

(b) 确定实现此移位寄存器所需模块（如图 6.1(a)所示）的最少个数。

(c) 在图 6.1(a)中标出左边模块的输入连接和内部路径。同时给出 X 和 Y 的函数。

6.4 一个由 2 个触发器（Q_1 和 Q_2）构成的时序电路，输入信号为 R, S 和 T，输出信号为 P，其下一状态方程为

$$D_1=Q_1^+=Q_2R+Q_1S$$
$$D_2=Q_2^+=Q_1+Q_2'T$$

输出方程为 $P=Q_2RT+Q_1ST$。

(a) 如何才能使用一个图 6.3 所示单片逻辑模块实现该时序电路？写出实现模块中每个函数生成器的方程。

(b) 在图 6.3 中标示出（高亮）输入信号、状态和输出变量及有效路径。

6.5 (a) 使用最少的逻辑模块（图 6.1(a)所示）实现一个 8 选 1 多路选择器，并给出每个模块中 X 和 Y 的函数，指出模块间的链接。

(b) 若使用图 6.3 所示模块，重复(a)的过程。给出每个模块中 X, Y 和 Z。

(c) 在(a)设计中 LUT 的内容为什么？

(d) 在(b)设计中 LUT 的内容为什么？

6.6 (a)写出图 6.1(a)中逻辑模块的 VHDL 代码，并使用下面的实体。

```
entity Figure6_1a is
  port(X_in, Y_in: in unsigned(1 to 4);
       clk, CE: in bit;
       Qx, Qy: out bit;
       X, Y: inout bit;
       XLUT, YLUT: in unsigned(0 to 15));
end Figure6_1a;
```

(b) 如果把两个 `Figure6_1a` 模块作为元件用于实现图 6.2 所示 4 选 1 多路选择器，请写出结构描述 VHDL 代码。当调用该模块元件时，使用 *XLUT* 和 *YLUT* 中存储的实际比特以设定每个 LUT 生成的函数。

6.7 (a) 写出图 6.3 中逻辑模块的 VHDL 代码。使用与习题 6.6(a)相似的实体部分，但是要增加 *ZLUT*, *SA*, *SB*, *SC* 和 *SD*，其中 *SC* 和 *SD* 为控制 4 个多路选择器的可编程选择位。在调用模块元件时，这些位均赋值为‘0’或‘1’。

(b) 把两个 `Figure6_3` 模块作为元件实现图 1.26 所示码转换器，写出其 VHDL 代码。当调用模块元件时，使用 *XLUT*, *YLUT* 和 *ZLUT* 中存储的实际比特设定每个 LUT 生成的函数。

6.8 (a) 实现 4 线-16 线译码器需要多少个图 6.1(a)所示逻辑模块？

(b) 给出第一个逻辑模块中 LUT 表的内容。

6.9 (a) 实现 8 线-3 线优先编码器需要多少个图 6.3 所示逻辑模块？

(b) 给出第一个逻辑模块中 LUT 表的内容。

6.10 说明用两个图 6.1(a)所示逻辑模块实现下面的组合逻辑函数。在图 6.1(a)中标示出连接情况，并给出每个模块中 X 和 Y 的函数。

$$F = X_1'X_2X_3'X_6 + X_2'X_3'X_4X_6' + X_2X_3'X_4' + X_2X_3X_4'X_6 + X_3'X_4X_5X_6' + X_7$$

6.11 应用图 6.1(a)所示逻辑模块写出下面的下一状态方程，要求使用的模块数最少。画出逻辑单元的连接图，并给出每一个单元中 X 和 Y 的函数（表达式已经是最简形式）。

$$Q^+ = UQV'W + U'Q'VX'Y' + UQX'Y + U'Q'V'Y + U'Q'XY + UQVW' + U'Q'V'X$$

6.12 实现下面函数，最少需要使用多少个图 6.3 所示逻辑模块？

$$X = X_1'X_2'X_3'X_4'X_5 + X_1X_2X_3X_4X_5 + X_5'X_6X_7'X_8'X_9 + X_5'X_6'X_7X_8X_9'$$

如果你的答案为 1，则在图 6.3 中指出所需的输入连接，并用粗线标示出内部连接路径。如果你的答案大于 1，则画出实现框图，要求标示出单元输入和各个单元的连接情况。不论答案为何，均要求给出实现每个 X, Y 和 Z 的函数生成器实现的函数。

6.13 已知 $Z(T,U,V,W,X,Y) = VW'X + U'V'WY + TV'WY'$。

(a) 说明使用一个图 6.3 所示逻辑模块实现 Z。在图 6.3 中画出逻辑单元的连接图，并给出每一个单元中 X 和 Y 的函数。

(b) 说明使用两个图 6.1(a)所示逻辑模块实现 Z。并画图说明每一个单元的输入，单元间的连接和每个单元中 X 和 Y 的函数。

6.14 应用香农展开定理对 a 和 b 展开下面的函数：

$$Y = abcde + cde'f + a'b'c'def + bcdef' + ab'cd'ef' + a'bc'de'f + abcd'e'f$$

以使其可以只使用 4 变量函数生成器就可实现。画出实现框图，并指出如何只使用一个 4 变量函数生成器就可以实现 Y。同时指出每个 4 变量函数生成器实现的函数。

6.15 应用香农展开定理，对 e 和 f 展开下面函数：

$$Y = ab'cdef + a'bc'd'e + b'c'ef' + abcde'f$$

以使其可以使用最少的 4 变量函数生成器即可实现。重写 Y，并指出如何只使用 4 变量函数生成器即可将其实现，画出实现框图。同时指出每个 4 变量函数生成器实现的函数。

6.16 (a) 应用香农展开定理，对 a 进行展开下面函数：

$$Y = ab'cd'e + a'bc'd'e + b'c'e + abcde$$

这样就可以使用 4 变量函数生成器将其实现。

(b) 对扩展函数加以解释，说明如何才能使用一个如图 6.3 所示单片逻辑模块实现此时序电路，并在图 6.3 中用高亮线标示出输入信号和有效路径。

(c) 给出 3 个 LUT 中的内容。

6.17 (a) 如果使用图 6.1(a)所示逻辑模块，则实现带累加器的 4 位加法器需要多少个 LUT?

(b) 如果使用图 6.11 中的 FPGA（内嵌进位链逻辑），需要使用多少个 4 输入 LUT?

(c) 设计一个 4 位加法/减法器，并使用带进位链逻辑模块和输入 LUT 的 FPGA 将其实现。假设控制信号 Su 在做加法运算时为 0，在做减法运算时为 1。在类似于图 6.11 的图上标出所需的连接，并给出每个 LUT 实现的函数。

6.18 用一个 FPGA 实现图 4.29 所示 4×4 数组乘法器。

(a) 把逻辑电路分区，并使用最小数量的图 6.1(a)所示逻辑模块。在一个逻辑模块中圈出每个组成元件集。确定需要 4 输入 LUT 的总数。

(b) 假设可以使用进位链逻辑，重复(a)中的操作。

6.19 (a) 对下面函数应用香农展开定理，要求对 A 进行展开，然后再在每个子函数中对 D 进行展开：

$$Z = AB'CD'E'F + A'BC'D'EF' + B'C'E'F + A'BC'E'F' + ABCDE$$

(b) 解释如何用两个 Xilinx Virtex FPGA 芯片（参见图 6.13）实现展开后的方程。在每个芯片图上，标示出 LUT（函数生成器）的输入，并画出连接路径，给出每个 LUT 实现的函数。

6.20 (a) 指出如何用图 6.15 中的开关链接实现下面的函数：

$$Z = AB'C + A'BC' + BC$$

(b) 指出如何用图 6.15 中开关链接实现下面的函数：

$$F = AB + A'C$$

(c) 指出如何用图 6.15 中开关链接实现锁存器（如图 2.17 所示）；

(d) 指出如何用图 6.15 中开关链接实现 D 触发器。

6.21 现有一个时序电路具有 5 个输入、2 个触发器和 2 个输出，其逻辑表达式为

$$Q_1^+ = Q_1(Q_2ABC) + Q_1'(Q_2'CDE)$$
$$Q_2^+ = Q_1'$$
$$Z_1 = Q_1'Q_2'AB + Q_1'Q_2'A'B' + Q_1Q_2'AB' + Q_1Q_2(A' + B + C)$$
$$Z_2 = Q_1A' + Q_1B + Q_2'$$

实现该逻辑表达式需要使用多少个 Virtex 逻辑片（参见图 6.13*）和触发器？具体给出每个逻辑片的输入和每个 LUT 实现的函数。

6.22 对现在市面上出售的 FPGA 芯片做一个调查。

(a) 根据调查的结果，给出一个与表 6.1 的相似表。

(b) 根据调查的结果，给出一个与表 6.2 的相似表。

6.23 如何用 4 个 16×16 乘法器和几个加法器实现 32×32 无符号数乘法？画出实现框图并标出所需连接。

6.24 用专用乘法器可以实现快速移位。左移 N 位等效于乘以 2^N。

(a) 已知 A 为一个 16 位无符号数，且 $0 \le N \le 15$。如何用一个乘法器和一个译码器实现一个向左移位器？

(b) 写出此类移位器的 VHDL 代码。

(c) 重复(a)和(b)的过程实现一个向右移位器。提示：乘以 2^{15-N}，并从 32 位积中选择恰当的 16 位。

6.25 对图 4.28(c)进行单热状态赋值，并给出下一状态方程和输出方程。

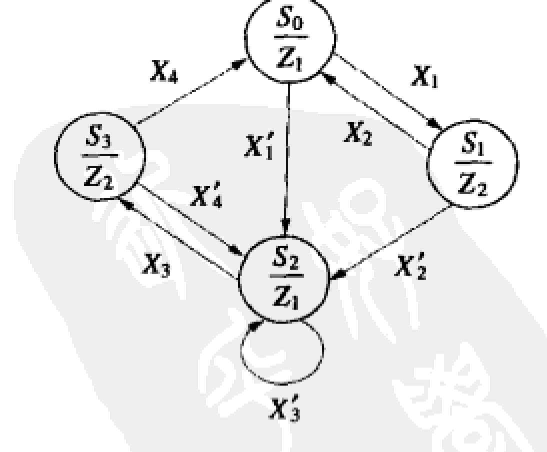
6.26 对图 4.53 进行单热状态赋值，并给出下一状态方程和输出方程。然后改变状态赋值，使 S_0 为 0000000，S_1 为 1100000，S_2 为 1010000，等等，并重新写出下一状态方程和输出方程。

6.27 假设使用单热状态赋值对一个时序系统的 4 个状态进行赋值，但是触发器无当前输入。触发器具有一个置位输入，因此其初始状态为 0000。如果采用单热状态赋值，则其他的状态应赋何值？给出解释。

6.28 对于所给的状态图，

(a) 直接写出化简的下一状态和输出方程。对触发器使用单热状态赋值，则 $Q_3Q_2Q_1Q_0$: S_0, 1000；S_1，0100；S_2, 0010；S_3, 0001。

(b) 实现这些方程需要多少个 Virtex 逻辑片（图 6.13）？



6.29 对交通灯控制器 VHDL 代码（图 4.15）做进一步的修正，使其用任何综合工具进行综合时不会生成锁存器（用 FPGA 或 CPLD 作为综合目标元件）。

6.30 综合二进制补码乘法器的行为描述模块（图 4.35），可以使用任何你有的综合工具。然后对图 4.40 的代码进行综合，并比较结果（触发器个数、LUT 个数、逻辑片的个数等）。对程序应用不同的综合选项（如：面积优化或速度优化）和不同的有限状态机编码算法（如单热赋值、紧凑赋值），并比较各个结果。如何设定综合选项才能使用最少的资源？

6.31 考虑下面的 VHDL 代码：

```
entity example is
  port(a: in integer range 0 to 3;
       b: out integer range 0 to 3);
end example;
architecture test2 of example is
begin
  process(a)
  begin
    case a is
      when 0 => b <= 3;
      when 1 => b <= 2;
      when 2 => b <= 1;
      when 3 => b <= 1;
    end case;
  end process;
end test2;
```

(a) 画出上述 VHDL 代码综合后（不进行优化）的硬件结构，并解释原因。

(b) 如果要使面积最小，应如何对硬件进行优化？给出优化步骤和理由。

6.32 画出下面 VHDL 代码综合后的硬件结构，其中 *A*, *B* 和 *E* 均为 4 位矢量，*C* 和 *D* 为 2 位数，*clock* 为 1 位信号。在画出的结构中标出输入和输出。

(a)
```
process(clock)
begin
  A <= A(3) & A(3 downto 1);
  B <= A(0) & B(3 downto 1);
end process;
```

(b)
```
architecture test2 of example is
begin
  process(C)
  begin
    case C is
      when 0 => D <= 3;
      when 1 => D <= 2;
      when 2 => D <= 0;
      when others => null;   -- preserver value
    end case;
  end process;
end test2;
```

(c)
```
architecture test2 of example is
begin
  process(C)
  begin
    case C is
      when 0 => E <= A + B;
      when 1 => E <= A sra 2;
      when 2 => E <= A - B;
      when 3 => E <= A;
    end case;
  end process;
end test2;
```

6.33 (a) 画出下面 VHDL 代码综合后的逻辑图（使用逻辑门、加法器、MUX 和 D 触发器等）。其中 *A*, *B* 和 *C* 均为无符号矢量（2 downto 0）。

```
process(CLK)
if CLK'event and CLK = 0 then
  if C0 = '1' then C <= not A; end if;
  if Ad = '1' then C <= A + B; end if;
  if Sh = '1' then C <= C sra 1; end if;
end if;
```

(b) 用一两句话说明该电路的用途。

6.34 写出下面 VHDL 代码综合后的典型硬件结构。如果代码中存在任何模糊之处，则指出你所做的假设。给出优化的和非优化的硬件。

(a)
```
architecture test2 of example is
begin
  process(a)
  begin
    case a is
      when 0 => b <= 2;
      when 1 => b <= 0;
      when 2 => b <= 3;
      when 3 => b <= 1;
    end case;
  end process;
end test2;
```

(b)
```
if arg1 > arg2 and arg1 > arg3 then
  result <= arg1;
else
  result <= '0';
end if;
```

6.35 假设 A 和 B 均为 8 位矢量，则语句

```
F <= (A>= B);
```

会导致何种硬件实现?

6.36 假设 A 是 4 位矢量，给出下列语句优化后的硬件实现：

```
F <= (A = 9);
```



第 7 章　浮点数算数

浮点数在计算机的程序中经常用于数值的计算。浮点数的计算单元通常比定点数的计算单元要复杂得多。浮点数既可以很大又可以很小。本章首先对浮点数进行简单的介绍，随后介绍了 IEEE 的浮点数规格，接着介绍了浮点数相乘的算法，并且用 VHDL 语言对其进行检验。然后完成了浮点数乘法器，并且用 FPGA 进行实现。最后对浮点数的加法、减法和除法也进行了简要的介绍。

7.1　浮点数的表示

一个浮点数（N）可以简单的用小数（F）、基数（B）和指数（E）来表示，即 $N = F \times B^E$。基数可以是 2, 10, 16 或任意其他数值。小数和指数可以用多种形式表示，例如，它们可以用二进制补码的形式表示，可以用有符号-绝对值的形式表示，也可以用其他数制表示。根据 F 和 E 允许的位数不同、基数不同以及 F 和 E 的负数表示不同，浮点数可以用多种不同的格式表示。基数既可以是隐式的，也可以是显式的。由于选择很多，所以在过去，浮点数的格式也有很多。

7.1.1　浮点数的二进制补码表示

本节中，我们介绍一种浮点数格式，我们用二进制补码表示浮点数的负指数和负小数，设指数的底（基数）为 2，因此浮点数可以表示为 $N = F \times 2^E$。在一个典型的浮点数系统中，F 的长度为 16～64 位，E 的长度为 8～15 位。为了使给出的例子简单且容易理解，我们使用 4 位小数和 4 位指数，但是这一概念很容易扩展到更多位。

这一数制系统的小数和指数都要用二进制补码表示（见 4.10 节中关于小数的补码表示）。我们使用 4 位小数和 4 位指数。小数部分包含符号位和 3 个实际小数位。默认二进制小数点在第一位之后。正数时符号位为 0，负数时符号位为 1。

下面我们把十进制数 2.5 用 8 位二进制补码浮点数格式表示。

$$\begin{aligned} 2.5 &= 0010.1000 \\ &= 1.010 \times 2^1 \quad \text{（规格化表示）} \\ &= 0.101 \times 2^2 \quad \text{（4 位二进制补码小数）} \end{aligned}$$

因此，

$$F = 0.101 \qquad E = 0010 \qquad N = 5/8 \times 2^2$$

如果为−2.5，那么指数不变，小数必须包含负号，所以二进制小数表示为 1.011。因此，

$$F = 1.011 \qquad E = 0010 \qquad N = -5/8 \times 2^2$$

下面我们再举一些例子，用 4 位小数和 4 位指数表示浮点数：

$$F = 0.101 \qquad E = 0101 \qquad N = 5/8 \times 2^5$$

$$F = 1.011 \qquad E = 1011 \qquad N = -5/8 \times 2^{-5}$$

$$F = 1.000 \qquad E = 1000 \qquad N = -1 \times 2^{-8}$$

为了利用 F 的所有位并使有效数字的位数最大，我们应该对 F 进行规格化，这样就可以使其绝对值尽可能的大。如果 F 不是规格化，我们可以通过左移 F，直到使其符号位和下一位不同为止，把 F 化为规格化。左移 F 等效于乘 2，所以为了保持绝对值 N 不变，我们每次左移 F 时都要对指数 E 减 1。进行规格化后，F 的绝对值将尽可能的扩大，因为如果再左移 F 的话，F 的符号就要发生改变了。下面例举的几个例子中，开始时 F 都不是规格化的，随后我们通过左移将其规格化。

非规格化：	$F = 0.0101$	$E = 0011$	$N = 5/16 \times 2^3 = 5/2$
规格化：	$F = 0.101$	$E = 0010$	$N = 5/8 \times 2^2 = 5/2$
非规格化：	$F = 1.11011$	$E = 1100$	$N = -5/32 \times 2^{-4} = -5 \times 2^{-9}$
（F 左移）：	$F = 1.1011$	$E = 1011$	$N = -5/16 \times 2^{-5} = -5 \times 2^{-9}$
规格化：	$F = 1.011$	$E = 1010$	$N = -5/8 \times 2^{-6} = -5 \times 2^{-9}$

指数可以是−8～+7 的任意数，小数可以为−1～+0.875 的任意数。

0 不能进行规格化，所以当 $N = 0$ 时，$F = 0.000$，指数可以为任意数；然而我们最好对数值 0 的表示形式做出统一规定。我们规定小数为 0，指数为负的最大值，由于在 4 位二进制补码的整数系统中，最大的负数是 1000，表示−8，所以当 F 和 E 均为 4 位时，0 可以表示为

$$F = 0.000 \qquad E = 1000 \qquad N = 0.000 \times 2^{-8}$$

一些浮点数系统使用偏置指数，这时，0 用 $F = 0$，$E = 0$ 联合表示。

7.1.2 IEEE 754 浮点数格式

IEEE 754 是 IEEE 在 1985 年建立的浮点数规格。它包含浮点数的两种表示格式：IEEE 单精度格式和 IEEE 双精度格式。IEEE 754 单精度格式用 32 位表示，双精度格式用 64 位表示。

虽然二进制补码通常用来表示负数，但是 IEEE 浮点表示法的小数和指数均不用二进制补码表示。IEEE 754 的设计者想要推出一套易于分类的格式，因此小数部分采用符号-绝对值计数法，指数采用偏置计数法。

IEEE 754 浮点格式有三个子字段：符号、小数和指数。小数部分用 IEEE 浮点格式规定的符号-绝对值计数法表示（例如小数部分含有显式符号位 S）。正数的符号位为 0，负数的符号位为 1。在二进制规格化科学计数法中，二进制小数点前的一位永远为 1，因此 IEEE 格式设计者决定把它设计成隐式的，只表示二进制小数点后的位。通常，在这种格式下浮点数可以表示为

$$N = (-1)^S \times (1 + F) \times 2^E$$

其中，S 是符号位；F 是小数部分；E 是指数；基数为 2。基数是隐式的（因为它没有在表示法的任何地方进行存储）。由于省略了起始位 1，所以此数的绝对值为 $1 + F$。有效数字是指小数的绝对值，在 IEEE 格式中，有效数字等于 $1 + F$。但是，在很多情况下，绝对值和小数这两个词是可以相互交换使用的，在本书中也是如此。

IEEE 浮点格式使用偏置计数法表示指数。偏置计数法是指任何数都可以表示成一个数加上一定的偏置值。在 IEEE 单精度格式中，偏置值为 127。因此，如果指数为+1，则应表示为+1+127 = 128；如果指数为−2，那么应表示为−2 + 127 = 125。因此当指数值小于 127 时，表示实际指数为负；当指数值大于 127 时，表示实际指数为正。在双精度格式中，偏置值为 1023。

如果一个正指数太大以至于指数字段放不下，那么就会产生向上溢出；如果一个负指数太大以至于指数字段放不下，那么就会产生向下溢出。

IEEE 单精度格式

IEEE 单精度格式使用 32 位来表示浮点数，它有三个子字段，见图 7.1 所示。第一个字段为小数部分的符号位。下个字段包含 8 位，用于指数。第三个字段有 23 位，用于小数部分。

S	指数	小数
1 位	8 位	23 位

图 7.1　IEEE 单精度浮点数格式

符号位反映小数的符号。正数的符号位为 0，负数的符号位为 1。当用 IEEE 单精度格式表示一个数时，首先要把这个数转化为规格化科学计数形式，在小数点前有且只有一位，并且同时调整指数值。

用 IEEE 754 标准格式表示指数时，要把实际指数值加上 127。表示规格化浮点数时，指数的范围为 1～254。指数值 0 和 255 保留，用于其他特殊用途，我们将在以后介绍。

要表示的数在用规格化科学计数表示后，除去第一位，我们就可以得到 23 位小数。0 不能用这种格式表示，因此我们可以对其进行特殊处理（在下文进行解释）。由于规格化科学计数中的每个数的第一位均为 1，所以可以把这个 1 去掉，这样有效数字又可以多一位了。这样，我们就可以用 24 位表示小数，而不是用 23 位表示小数了。IEEE 格式的设计者就是想最大限度地利用指数和小数字段的所有位。

为了更好地理解 IEEE 格式，下面我们用 IEEE 浮点数格式表示 13.45。由于 0.45 是一个无限循环的二进制小数，所以 13.45 = 1101.01 1100 1100 1100 ... ... 1100 是其无限循环位。

若用规格化科学表示法表示，则

$$13.45 = 1.10101\ 1100\ 1100.... \times 2^3$$

由于此数为正数，所以用 IEEE 754 格式表示时，其符号位为 0。

指数用偏置计数法表示为 127 + 3 = 130，用二进制表示为 10000010。

小数部分为 1.10101 1100 1100 ... ...（1100 无限循环）。把第一位的 1 省略，用 23 位表示小数部分为

$$10101\ 1100\ 1100\ 1100\ 1100\ 11$$

这样，完整的 32 位表示为

$$0\ 10000010\ 10101\ 1100\ 1100\ 1100\ 1100\ 11$$

如图 7.2 所示。

S	指　数	小　数
0	10000010	10101110011001100110011

图 7.2　13.45 的 IEEE 单精度浮点数表示

为了方便起见，上面的 32 位数可以表示为十六进制格式：

$$4157\ 3333$$

若要表示−13.45，则只需要把符号位从 0 变为 1。因此−13.45 用 IEEE 754 单精度十六进制格式表示为 C157 3333。

IEEE 双精度格式

IEEE 双精度格式使用 64 位数表示浮点数，如图 7.3 所示。第一位为小数部分的符号位。紧接着的 11 位用来表示指数，其他的 52 位表示小数部分。

S	指 数	小 数
1 位	11 位	52 位

图 7.3 IEEE 双精度浮点数格式

同单精度格式相同，正数的符号位为 0，负数的符号位为 1。

第二个字段用来表示指数部分，在表示时要把规格化的实际指数值加上 1023。在规格化浮点数格式中指数的取值范围为 1～2046。指数值 0 和 2047 保留，作为其他特定用途。

我们用规格化科学计数表示浮点数，并省略其小数部分的第一位 1，然后只把接下来的 52 位作为格式中的 52 位小数部分。

下面我们用 IEEE 双精度浮点数格式表示 13.45。把 13.45 用转换为二进制表示。

$$13.45 = 1101.01\ 1100\ 1100\ 1100 \ldots\ldots \quad (1100\text{ 无限循环})$$

用规格化科学计数法表示为

$$13.45 = 1.10101\ 1100\ 1100 \ldots \times 2^3$$

指数用偏置记数法表示为 1023 + 3 = 1026，用二进制表示为

$$10000000010$$

小数部分为 1.10101 1100 1100 ... ...（1100 无限循环）。把第一位的 1 省略，用 52 位表示小数部分为

$$10101\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 110$$

这样，完整的 64 位表示为

$$0\ 10000000010\ 10101\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 110$$

如图 7.4 所示。为了方便起见，上面的 64 位数用十六进制（hex）格式表示为

$$402A\ E666\ 6666\ 6666$$

若要表示−13.45，则只需要把符号位从 0 变为 1。因此−13.45 用 IEEE 754 双精度十六进制格式表示为 C02A E666 6666 6666。

S	指 数	小 数
0	10000010	1010111001100110011001100110011001100110011001100110

图 7.4 13.45 的 IEEE 双精度浮点数表示

IEEE754 标准中的特殊表示

IEEE754 标准中有几个特殊表示，如图 7.5 所示。这些特殊表示包括 0、无穷、非规格化数和

NaN（Not a Number，非数值）。我们用最大和最小指数表示这些特殊数字。

单精度		双精度		表示
指数	小数	指数	小数	
0	0	0	0	0
0	非零	0	非零	±非规格化数
255	0	2047	0	± 无穷
255	非零	2047	非零	NaN

图 7.5　IEEE 754 浮点数格式中的特殊表示

零　在 IEEE 格式中，我们用指数和小数部分的全 0 来表示数值 0。之所以把 0 作为特殊处理，是因为基于规格化格式很难表示 0。把 IEEE 格式的规格化数进行转化时，我们经常把小数部分加 1，但是对于 0，这是不可能的。

非规格化数　单精度格式可以表示的最小规格化数为

$$1.0 \times 2^{-126}$$

从 1.0×2^{-126}～0 之间的所有数都不可以用规格化格式表示。如果在格式中对数的规格化并不是必须的，则我们可以表示小于 1.0×2^{-126} 的了。因此 IEEE 浮点数格式把这些不可以被规格化的数（非规格化数）作为特殊处理。如果一个数的指数为 0，小数部分不为 0，则这个数为非规格化数。现在，IEEE 格式可以表示的最小数为

$$0.00000000000000000000001 \times 2^{-126}\text{，即 } 1.0 \times 2^{-149}$$

这样，对非规格化数的定义，可以表示 1.0×2^{-149}～1.0×2^{-126} 之间的数了。

对于双精度格式来说，对非规格化数的定义，可以表示 1.0×2^{-1074}～1.0×2^{-1022} 之间的数了。

无穷　在表示无穷时，我们令指数达到最大值，小数部分为 0。在单精度格式下，指数值为 255；在双精度格式下，指数值为 2047。

NaN　IEEE 754 标准把无效操作(例如 0/0)的结果作为特殊处理，并把这些结果表示为 NaN。当指数为 255，小数为任意非零数时，我们就认为这个数为 NaN（非数值）。

舍入　当现有的数位比所需的数位少时，就会涉及到舍入问题。我们期望舍入后的值可以最贴近原值。当原值大于中间值时，只入不舍；当原值小于中间值时，只舍不入。另一种选择是截尾，即把超出允许位数的位舍去。为了得到较高的精确度，我们必须在格式中间再加入几位。为了更好地舍入，IEEE 标准在数制表示格式中又加入了两位，我们称这两位为保护位和舍入位。有时，除了保护位和舍入位外，还要使用第三个附加位，我们称之为捆绑位。当舍入位的右边一位不为 0 时，就要使用捆绑位。

当原数值正好等于中间值时是最难处理的。IEEE 标准有下面四种不同的舍入模式：

- **只入不舍**　向正无穷方向进行舍入，只入不舍。最终结果为下一个大于原值的数值。
- **只舍不入**　向负无穷方向进行舍入，只舍不入。最终结果为前一个小于原值的数值。
- **截尾**　向 0 方向进行舍入。把超出允许位数的位忽略。与符号-绝对值计数法的截尾舍入相同。
- **去偏置**　如果原数值正好等于中间值，那么有时只入不舍，有时只舍不入。如果被保留的最低位为 1，则加 1，且只入不舍；如果为 0，则截尾。此种舍入结构会造成舍入后数值的最低保留位恒为 0。

7.2 浮点数乘法

已知两个浮点数 $F_1 \times 2^{E_1}$ 和 $F_2 \times 2^{E_2}$，它们的积为

$$\left(F_1 \times 2^{E_1}\right) \times \left(F_2 \times 2^{E_2}\right) = (F_1 \times F_2) \times 2^{(E_1+E_2)} = F \times 2^E$$

积的小数部分是两个数的小数部分的积，指数部分是两个指数的和。因此，浮点数乘法器包括两个主要组成部分：小数乘法器和指数加法器。浮点数乘法的表示具体取决于所选用的小数乘法和指数加法的精确格式。

我们可以通过很多方法计算小数乘法。如果使用 IEEE 标准格式小数，那么要先计算积的绝对值，然后确定积的符号。如果使用二进制补码格式的小数，那么我们可以直接使用能够计算带符号二进制补码的乘法器，这种乘法器我们在第 4 章已经讨论过了。

指数的加法操作可以用二进制加法器进行。如果直接使用 IEEE 格式指数进行加法运算，那么为了得到正确的结果就必须对和的表示形式加以调整。例如：当两个 IEEE 格式的指数相加后，得到的和中包含 2 倍的偏置值，为了使结果正确，我们必须从和中减去一个偏置值。

二进制补码系统在进行算术运算时，有很多有趣的性质。因此，很多浮点数算术单元都先把 IEEE 格式的数转化为二进制补码形式，然后在内部使用二进制补码进行浮点数运算，当得到最终结果时，再把结果转化为 IEEE 格式。

浮点数乘法的基本实现步骤如下所示：

1. 计算两个指数的和。
2. 计算两个小数（有效数字）的积。
3. 如果积为 0，则调整表示形式，并正确的表示结果（0 为 IEEE 格式中的特例）。
4. a. 如果积的值太大，那么通过对其进行右移进而实现规格化，同时增加指数的值。
   b. 如果积的值太小，那么通过对其进行左移进而实现规格化，同时减小指数的值。
5. 如果指数向下或向上溢出，则生成一个异常或错误指示。
6. 对结果的位数进行恰当的舍入。如果由于舍入导致结果不是规格化的，那么再次回到第 4 步。

注意，除了要计算指数的和及小数的积，我们还需要进行很多其他步骤，例如：对积进行规格化、处理向下或向上溢出、对结果的位数进行恰当的舍入，等等。我们假设开始时，两个数都是规格化的，所以我们希望最终结果也是规格化的。

现在，我们讨论如何设计浮点数乘法器。假设小数为 4 位，指数也为 4 位，负数均用二进制补码表示。

基本上，运算中我们要做的就是指数的相加（步骤 1）和小数的相乘（步骤 2）。然而，我们必须考虑到一些特殊的情况。首先，如果 F 是 0，那么我们必须令指数 E 为最大的负值(1000)（步骤 3）。第二，如果是$-1 \times -1(1.000 \times 1.000)$，则结果应是+1。由于我们无法把+1 表示成具有 4 位小数的二进制补码形式，所以对于这种特殊情况，我们要把结果右移，也就是步骤 4。为了纠正这种情况，我们令 $F = 1/2(0.100)$，并把 E 加 1。这时计算结果是正确的，因为 $1 \times 2^E = 1/2 \times 2^{E+1}$。

当我们进行小数乘法运算时，结果可以不是规格化的。例如，

$$\left(0.1 \times 2^{E_1}\right) \times \left(0.1 \times 2^{E_2}\right) = 0.01 \times 2^{(E_1+E_2)} = 0.1 \times 2^{(E_1+E_2-1)}$$

这就是步骤 4.b 所说的情况。在这种情况下，我们通过使小数左移一位，且指数减 1 来完成结果的规格化。最后，如果指数的运算结果太大而不能在这种数值系统中表示出来时，就产生了指数的向上溢出（负方向上的向上溢出就是向下溢出）。由于我们用的是 4 位的指数，所以如果指数不在 1000～0111（−8～+7）的范围内，就会产生向上溢出。因为这种指数的溢出是不能被纠正的，所以这时溢出指示器开启（步骤 5）。

浮点数乘法的流程图见图 7.6 所示。在小数乘法运算完成后，所有的特殊情况都被检测出来。因为 F_1 和 F_2 是规格化的小数，所以运算后积的最小可能数值是 0.01，正如上面例子中提到的。因此，只要左移一位就可以使 F 变为规格化小数。

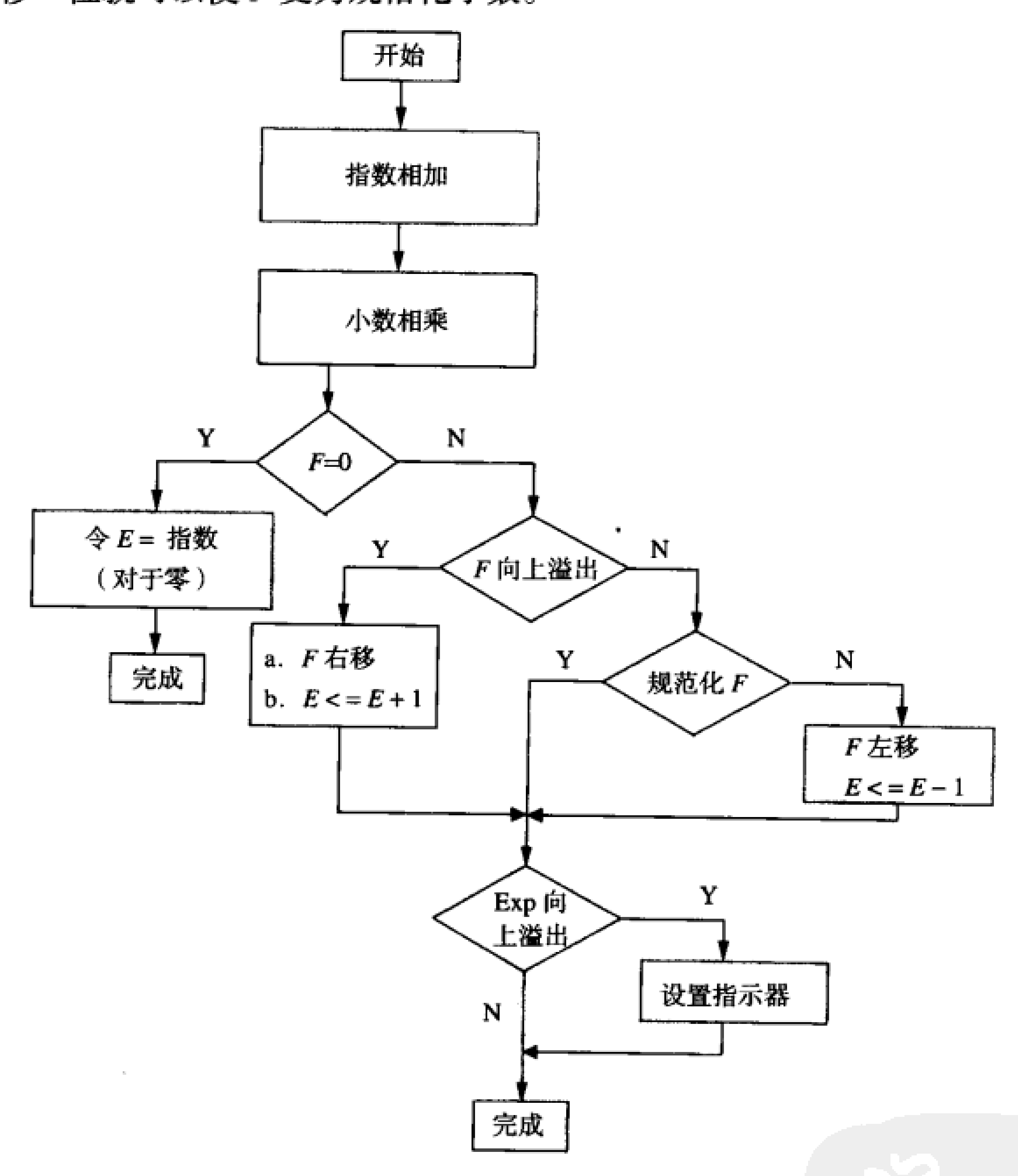


图 7.6　浮点数乘法（指数和小数用二进制补码表示）

实现乘法器（见图 7.7）的硬件必须包括一个指数加法器、一个小数乘法器和一个控制单元。控制单元为实现正确的左移、右移和加减等操作提供信号。

指数加法器　由于二进制补码加法操作可以得到正确格式的和，所以指数加法器就可以很直观的设计出来。我们用一个 5 位全加器作为指数加法器，如图 7.7 所示。当小数规格化后，指数将做出相应的增加或减少。在特殊情况下，即积为 0 时，寄存器应该设置为 1000。寄存器上的控制信号可以控制寄存器进行增加或减少，直至设置为负数最大值（SM8）。

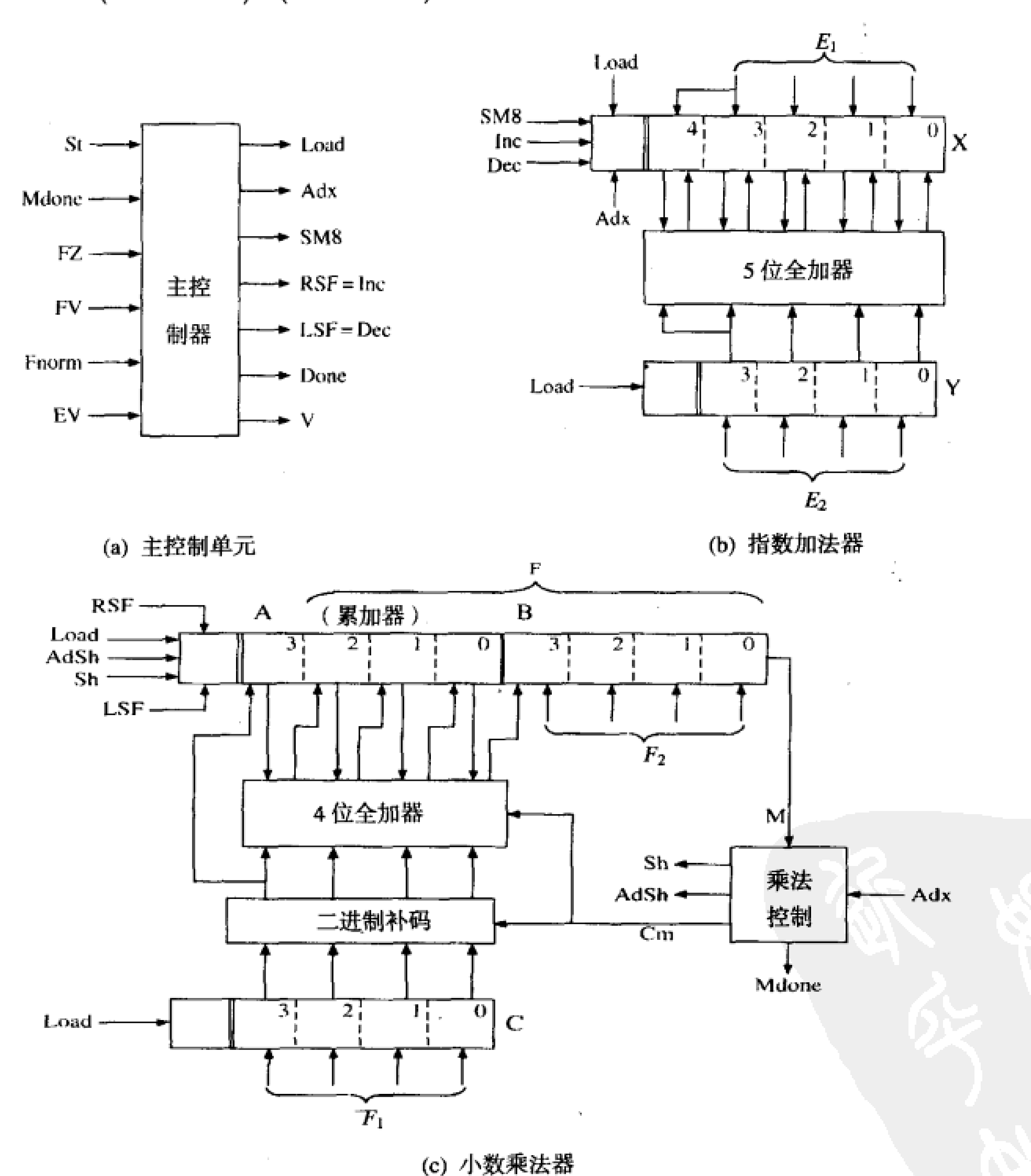
为了解决特殊情况，我们使用一个 5 位寄存器来保存计算结果 sum。当进行指数加法运算时，会发生向上溢出。当 E_1 和 E_2 均为正数，而二者的和（E）为负数，或者当 E_1 和 E_2 均为负数时，

而二者的和为正数，这种结果就叫做二进制补码的向上溢出。但是，这种向上溢出可能可以通过规格化过程中对 E 加 1 或减 1 而进行纠正，或者通过小数向上溢出校正进行纠正。为了包容这种情况，我们使用了长度为 5 位的寄存器 X。当 E_1 的值被载入到 X 中时，一定要对符号位进行扩展，这样我们就能够用正确的二进制补码形式进行表示。由于存在两个符号位，所以当相加产生向上溢出时，符号位的低位将发生改变，但是符号位的高位不会发生变化。下面的两个例子都产生了向上溢出，因此符号位的低位都是错误的值：

$$7 + 6 = 00111 + 00110 = 01101 = 13 \quad \text{（最大允许值是 7）}$$
$$-7 + (-6) = 11001 + 11010 = 10011 = -13 \quad \text{（最大允许负值是-8）}$$

下面的例子给出了一种特殊的情况：当小数和指数同时发生向上溢出时，通过纠正小数的向上溢出进而纠正了指数的向上溢出。

$$\left(1.000\times2^{-3}\right)\times\left(1.000\times2^{-6}\right)=01.000000\times2^{-9}=00.100000\times2^{-8}$$



(a) 主控制单元

(b) 指数加法器

(c) 小数乘法器

图 7.7 浮点数乘法的主要组成部分

小数乘法器 我们在 4.10 节介绍的小数乘法器为实现二进制补码乘法运算提供了一个直观

的方法。因此，我们对其进行修改使其变为浮点数乘法器。它提出了一种移位相加的乘法器实现算法。由于我们要计算 3 位（再加上符号）乘以 3 位（再加上符号）的乘法运算，所以得到的积为 6 位（再加上符号位）。因此在小数乘法计算完毕时，得到的结果为 7 位（F），由 A 的低三位与 B 拼接。此乘法器具有自己的控制单元，可以根据乘数位生成恰当的移位和相加信号。

主控制单元 浮点数乘法器主控制器的 SM 图（图 7.8）是基于流程图的。我们把它叫做主控制器是为了跟乘法器的控制器相区别。乘法器的控制器是一个单独的状态机，并与主控制器相连。

SM 图使用的输入和控制信号如下所示：

St	开始进行浮点数的乘法运算
Mdone	小数的相乘运算结束
FZ	小数为 0
FV	小数值溢出（小数值太大）
Fnorm	对 F 进行规格化
EV	指数向上溢出
Load	把 F_1，E_1，F_2，E_2 的值载入到正确的寄存器中（在准备进行乘法运算的同时清除 A 的值）
Adx	指数相加；此信号还启动小数乘法运算
SM8	令指数的值为−8（解决为 0 的特殊情况）
RSF	小数右移一位；同时对 E 加 1
LSF	小数左移一位；同时对 E 减 1
V	向上溢出指示器
Done	浮点数的乘法运算完成

主控制器的 SM 图有 4 个状态。在 S_0 状态，当开始信号为 1 时，寄存器载入数值。在 S_1 状态，开始进行指数的加法运算和小数的乘法运算。在 S_2 状态，我们等待小数的乘法运算的结束，然后开始检测特殊情况是否存在，并且进行适当的处理。可能会令人感到惊讶的是虽然 *FZ*，*FV* 和 *Fnorm* 在流程图中是顺序执行的，但是我们几乎在同一个状态内完成它们的检测。然而，*FZ*，*FV* 和 *Fnorm* 是由并行执行的组合电路产生的，因此它们可以在同一个状态内检测。但是，我们必须等到指数在下一个时钟完成加 1 或者减 1 后，才能在 S_3 状态内检测指数是否存在向上溢出。在 S_3 状态，结束信号（*Done*）启动，直到控制器等到信号 $St = 0$ 时，回到 S_0 状态。

乘法器的控制状态图（图 7.9）与图 4.34 很相似，但是由于寄存器载入的是主控制器的信息，所以不需要载入状态。相加和移位操作在一个状态内进行，因为如图 7.7(c)所示，加法器的和线在载入到累加寄存器前移了一位。当 Adx =1 时，乘法器开始工作，当乘法运算完成时，*Mdone* 启动。

VHDL 行为描述模块（见图 7.10）使用了 3 个进程。主进程根据 SM 表生成控制信号。第二个进程为小数乘法器生成控制信号。第三个进程对控制信号进行检测，并且在时钟上升沿对相应的寄存器进行更新。在主进程的状态 S_2 中，A = “0000”就意味着 F=0（在 SM 表中 $FZ = 1$）。如果进行乘法运算 1.000×1.000，则结果为 $A\&B$ = “01000000”，并且产生小数向上溢出($FV = 1$)。如果 $A(2)=A(1)$，则 F 的符号位与其下一位相同，这样 F 就不可以进行规格化($Fnorm$=0)。在状态 S_3 中，如果 X 的最高两位不同，则产生指数向上溢出($EV = 1$)。

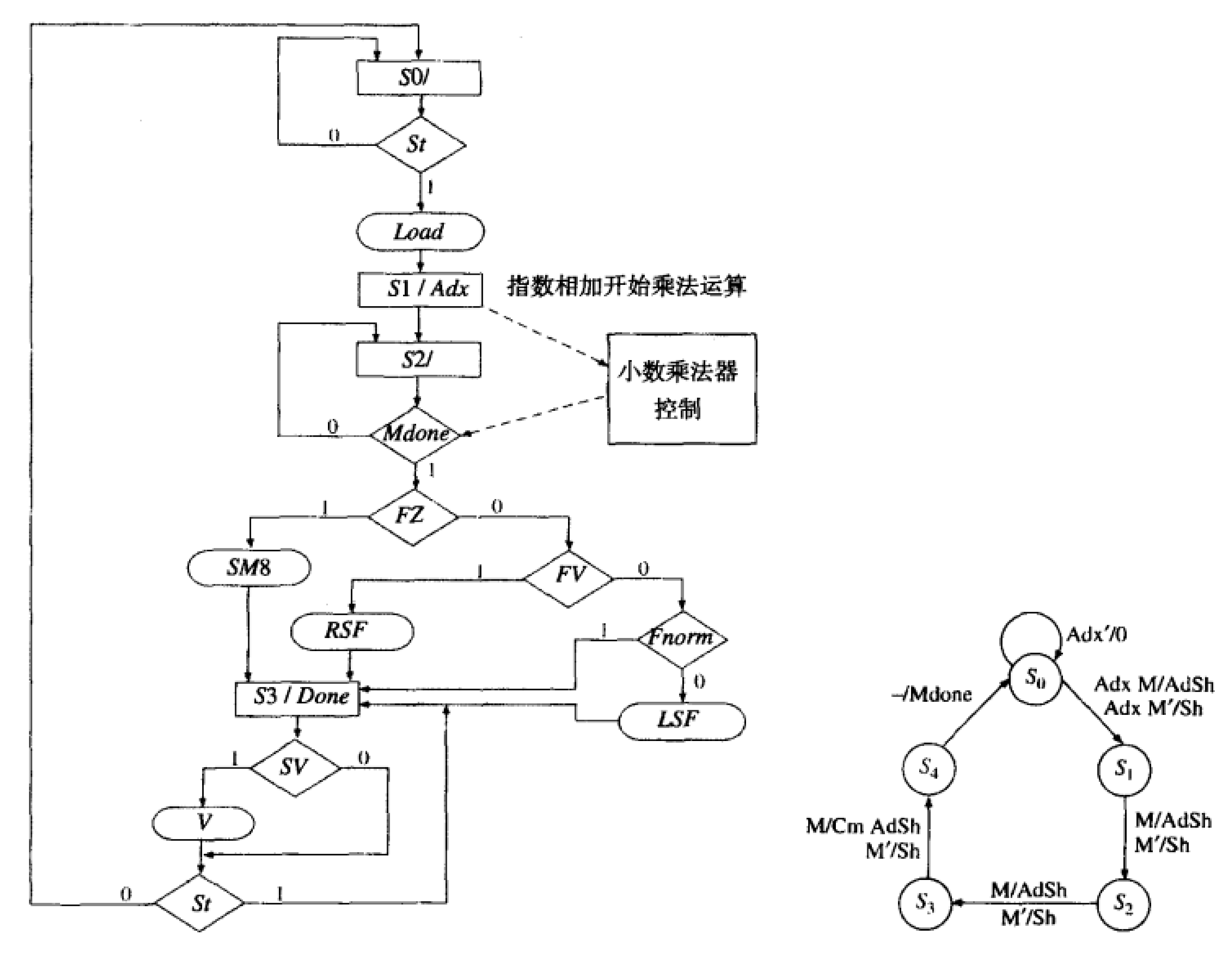


图 7.8 浮点数乘法的 SM 图

图 7.9 乘法器控制的状态图

在第三个进程中，寄存器进行更新。变量 *addout* 表示 4 位全加器的输出，它是小数乘法器的一部分。当 *Cm*=1 时，此加法器把二进制补码形式的 *C* 与 *A* 相加。当 *Load*=1 时，符号扩展后的指数被载入到 *X* 和 *Y* 中。当 *Adx*=1 时，向量 *X* 和 *Y* 相加。当 *SM8*=1 时，-8 被载入到 *X* 中。当 *Adsh*=1 时，*A* 连同 *C* 的符号位（当 *Cm*=1 时，为符号位的补码）一起被载入，并与加法器的输出矢量（4 位）拼接，*addout* 中余下的位移到寄存器 B 中。

```
library IEEE;
use IEEE.numeric_bit.all;

entity FMUL is
  port(CLK, St: in bit;
       F1, E1, F2, E2: in unsigned(3 downto 0);
       F: out unsigned(6 downto 0);
       V, done: out bit);
end FMUL;
architecture FMULB of FMUL is
signal A, B, C: unsigned(3 downto 0);           -- fraction registers
signal X, Y: unsigned(4 downto 0);              -- exponent registers
signal Load, Adx, SM8, RSF, LSF: bit;
signal AdSh, Sh, Cm, Mdone: bit;
signal PS1, NS1: integer range 0 to 3;          -- present and next state
signal State, Nextstate: integer range 0 to 4; -- multiplier control state
begin
  main_control: process(PS1, St, Mdone, X, A, B)
  begin
    Load <= '0'; Adx <= '0'; NS1 <= 0;          -- clear control signals
    SM8 <= '0'; RSF <= '0'; LSF <= '0'; V <= '0'; F <= "0000000";
    done <= '0';
    case PS1 is
```

图 7.10 浮点数乘法的 VHDL 代码

```
      when 0 => F <= "0000000";                  -- clear outputs
        done <= '0'; V <= '0';
        if St = '1' then Load <= '1'; NS1 <= 1; end if;
      when 1 => Adx <= '1'; NS1 <= 2;
      when 2 =>
        if Mdone = '1' then                      -- wait for multiply
          if A = 0 then                          -- zero fraction
            SM8 <= '1';
          elsif A = 4 and B = 0 then
            RSF <= '1';                          -- shift AB right
          elsif A(2) = A(1) then                 -- test for unnormalized
            LSF <= '1';                          -- shift AB left
          end if;
          NS1 <= 3;
        else
          NS1 <= 2;
        end if;
      when 3 =>                                  -- test for exp overflow
        if X(4) /= X(3) then V <= '1'; else V <= '0'; end if;
        done <= '1';
        F <= A(2 downto 0) & B;                  -- output fraction
        if ST = '0' then NS1 <= 0; end if;
    end case;
  end process main_control;

  mul2c: process(State, Adx, B)                  -- 2's complement multiply
  begin
    AdSh <= '0'; Sh <= '0'; Cm <= '0'; Mdone <= '0';   -- clear control signals
    Nextstate <= 0;
    case State is
      when 0 =>                                   -- start multiply
        if Adx = '1' then
          if B(0) = '1' then AdSh <= '1'; else Sh <= '1'; end if;
          Nextstate <= 1;
        end if;
      when 1 | 2 =>                               -- add/shift state
        if B(0) = '1' then AdSh <= '1'; else Sh <= '1'; end if;
        Nextstate <= State + 1;
      when 3 =>
        if B(0) = '1' then Cm <= '1'; AdSh <= '1'; else Sh <= '1'; end if;
        Nextstate <= 4;
      when 4 =>
        Mdone <= '1'; Nextstate <= 0;
    end case;
  end process mul2c;

  update: process                                 -- update registers
  variable addout: unsigned(3 downto 0);
  begin
    wait until CLK = '1' and CLK'event;
    PS1 <= NS1;
    State <= Nextstate;
    if Cm = '0' then addout := A + C;
    else addout := A - C;
    end if;                                       -- add 2's comp. of C
    if Load = '1' then
      X <= E1(3) & E1; Y <= E2(3) & E2;
      A <= "0000"; B <= F1; C <= F2;
    end if;
    if ADX = '1' then X <= X + Y; end if;
    if SM8 = '1' then X <= "11000"; end if;
    if RSF = '1' then A <= '0' & A(3 downto 1);
      B <= A(0) & B(3 downto 1);
      X <= X + 1;
    end if;                                       -- increment X
    if LSF = '1' then
      A <= A(2 downto 0) & B(3); B <= B(2 downto 0) & '0';
      X <= X + 31;
    end if;                                       -- decrement X
    if AdSh = '1' then
      A <= (C(3) xor Cm) & addout(3 downto 1);    -- load shifted adder
      B <= addout(0) & B(3 downto 1);             -- output into A & B
    end if;
    if Sh = '1' then
      A <= A(3) & A(3 downto 1);                  -- right shift A & B
      B <= A(0) & B(3 downto 1);                  -- with sign extend
    end if;
  end process update;
end FMULB;
```

图 7.10（续）　浮点数乘法的 VHDL 代码

我们应该对浮点数乘法器的 VHDL 代码进行认真的测试，此测试应包含所有的特例，包括正、负小数和正、负指数。图 7.11 给出了指令文件和一些检测结果。这并不是一个完备的检测。

```
add list f x f1 e1 f2 e2 v done
force f1 0111 0, 1001 200, 1000 400, 0000 600, 0111 800
force e1 0001 0, 1001 200, 0111 400, 1000 600, 0111 800
force f2 0111 0, 1001 200, 1000 400, 0000 600, 1001 800
force e2 1000 0, 0001 200, 1001 400, 1000 600, 0001 800
force st 1 0, 0 20, 1 200, 0 220, 1 400, 0 420, 1 600, 0 620, 1 800, 0 820
force clk 0 0, 1 10 -repeat 20
run 1000

ns  delta    f        x     f1   e1   f2   e2  v done
  0   +0  0000000  00000 0000 0000 0000 0000 0 0
  0   +1  0000000  00000 0111 0001 0111 1000 0 0    (0.111 × 2^1) × (0.111 × 2^-8)
 10   +1  0000000  00001 0111 0001 0111 1000 0 0
 30   +1  0000000  11001 0111 0001 0111 1000 0 0
150   +2  0110001  11001 0111 0001 0111 1000 0 1    = 0.110001 × 2^-7
170   +2  0000000  11001 0111 0001 0111 1000 0 0
200   +0  0000000  11001 1001 1001 1001 0001 0 0    (1.001 × 2^-7) × (1.001 × 2^1)
250   +1  0000000  11010 1001 1001 1001 0001 0 0
370   +2  0110001  11010 1001 1001 1001 0001 0 1    = 0.110001 × 2^-6
390   +2  0000000  11010 1001 1001 1001 0001 0 0
400   +0  0000000  11010 1000 0111 1000 1001 0 0    (1.000 × 2^7) × (1.000 × 2^-7)
430   +1  0000000  00111 1000 0111 1000 1001 0 0
450   +1  0000000  00000 1000 0111 1000 1001 0 0
570   +1  0000000  00001 1000 0111 1000 1001 0 0
570   +2  0100000  00001 1000 0111 1000 1001 0 1    = 0.100000 × 2^1
590   +2  0000000  00001 1000 0111 1000 1001 0 0
600   +0  0000000  00001 0000 1000 0000 1000 0 0    (0.000 × 2^-8) × (0.000 × 2^-8)
630   +1  0000000  11000 0000 1000 0000 1000 0 0
650   +1  0000000  10000 0000 1000 0000 1000 0 0
770   +1  0000000  11000 0000 1000 0000 1000 0 0
770   +2  0000000  11000 0000 1000 0000 1000 0 1    = 0.0000000 × 2^-8
790   +2  0000000  11000 0000 1000 0000 1000 0 0
800   +0  0000000  11000 0111 0111 1001 0001 0 0    (0.111 × 2^7) × (1.001 × 2^1)
830   +1  0000000  00111 0111 0111 1001 0001 0 0
850   +1  0000000  01000 0111 0111 1001 0001 0 0
970   +2  1001111  01000 0111 0111 1001 0001 1 1    = 1.001111 × 2^8 (overflow)
990   +2  0000000  01000 0111 0111 1001 0001 0 0
```

图 7.11　浮点数乘法器的检测数据和仿真结果

当使用Xilinx ISE工具在Xilinx Spartan-3/Virtex-4的结构上对此VHDL代码进行综合时，我们得到的综合结果为38个片、29个触发器、72个4输入LUT、27个I/O模块和1个全局时钟电路。为了消除不必要的锁存器，输出信号*V*，*Done*和*F*在进程初始化时均设置为0。此时RTL层面的设计也已经完成，但是RTL设计并不比行为描述方式的设计高级。

现在基本的设计已经完成，我们需要确定浮点数乘法器的工作速度，它决定了乘法器的最大时钟频率。大多数 CAD 工具都提供了模拟最终电路的方法，此方法对电路的模拟已经包含逻辑模块延迟和连接延时。如果时钟分析显示设计的操作因不够快而不符合规格，那么我们可以进行一些其他的选择。大多数的 FPGA 都具有一些不同的速度等级，所以我们可以选择一个速度更快的部分。另一种方法是在电路中找出最长的时间延时路径，并且尝试重新进行链接或者重新设计电路以减小延时。

7.3 浮点数加法

接下来，我们考虑浮点数加法器的设计。两个浮点数相加，和也为浮点数：

$$(F_1 \times 2^{E_1}) + (F_2 \times 2^{E_2}) = F \times 2^E$$

我们认为进行加法运算的两个浮点数都是规格化的，并且结果也应该是规格化格式。为了使两个小数相加，这两个浮点数的指数必须是相等的。因此，如果指数 E_1 和 E_2 不同的话，我们必须通过使其中一个小数非规格化以实现指数的相等。应该调整较小的数，这样一来即使是重要的数字丢失，也不会产生很大的影响。为了展示此过程，我们把下面的两个数相加：

$$F_1 \times 2^{E_1} = 0.111 \times 2^5 \quad 和 \quad F_2 \times 2^{E_2} = 0.101 \times 2^3$$

由于 E_1 不等于 E_2，所以我们通过右移两次使 F_2 非规格化，同时对其指数加 2，可以得到

$$0.101 \times 2^3 = 0.0101 \times 2^4 = 0.00101 \times 2^5$$

我们注意到右移一位等价于除以 2，所以每当我们进行移位操作时，都可以通过对指数加 1 或者减 1 来抵消对结果的影响。当指数相等时，我们把两个小数相加：

$$(0.111 \times 2^5) + (0.00101 \times 2^5) = 01.00001 \times 2^5$$

此加法操作产生了一个溢出，此溢出进入到符号位，所以我们右移一位并对指数加 1，以此纠正小数的向上溢出。最终结果为

$$F \times 2^E = 0.100001 \times 2^6$$

当其中的一个小数是负时，小数相加的结果可能是非规格化的，如下例所示：

$$\begin{aligned}
&(1.100 \times 2^{-2}) + (0.100 \times 2^{-1}) \\
&= (1.110 \times 2^{-1}) + (0.100 \times 2^{-1}) \quad （F_1 移位后） \\
&= 0.010 \times 2^{-1}（相加的结果是非规格化的） \\
&= 0.100 \times 2^{-2}（通过左移一位的同时对指数减 1 来实现结果的规格化）
\end{aligned}$$

总之，浮点数加法的计算步骤如下：

1. 比较指数值。如果指数不等，则将指数较小的那个浮点数的小数向右移一位，并将其指数加 1。重复操作直到两个指数相等为止。
2. 对两个小数进行加法运算。
3. 如果结果为 0，则把指数设置成恰当的表示形式，并输出。
4. 如果出现小数的向上溢出，则对其右移一位同时对指数加 1 来纠正溢出。
5. 如果小数是非规格化的，则对其左移一位同时对指数减 1 来实现规格化。
6. 检查是否存在指数的向上溢出，如果有，则启动溢出指示器。
7. 对位数进行适当的舍入。是否仍为规格化的数？如果不是，则返回步骤 4。

图 7.12 用流程图的方式给出了上述步骤。在步骤 1 中可以加入优化进程。我们可以首先断定这两个数是否相差很大。如果 $E_1 \gg E_2$，并且 F_2 为正，那么如果通过右移 F_2 来使两个指数相等，则可能会造成 F_2 全为 0。此时，结果为 $F=F_1$，$E=E_1$，所以移位完全是浪费时间。如果 $E_1 \gg E_2$，并且 F_2 为负，那么如果通过右移 F_2 来使两个指数相等，则可能会造成 F_2 全为 1，当进行加法运算时就会得到错误的结果。为了避免这个问题，当 $E_1 \gg E_2$ 时，我们可以跳过移位这一步，直接设定 $F=F_1$，$E=E_1$。同理，如果 $E_2 \gg E_1$，我们也可以跳过移位直接设定 $F=F_2$，$E=E_2$。

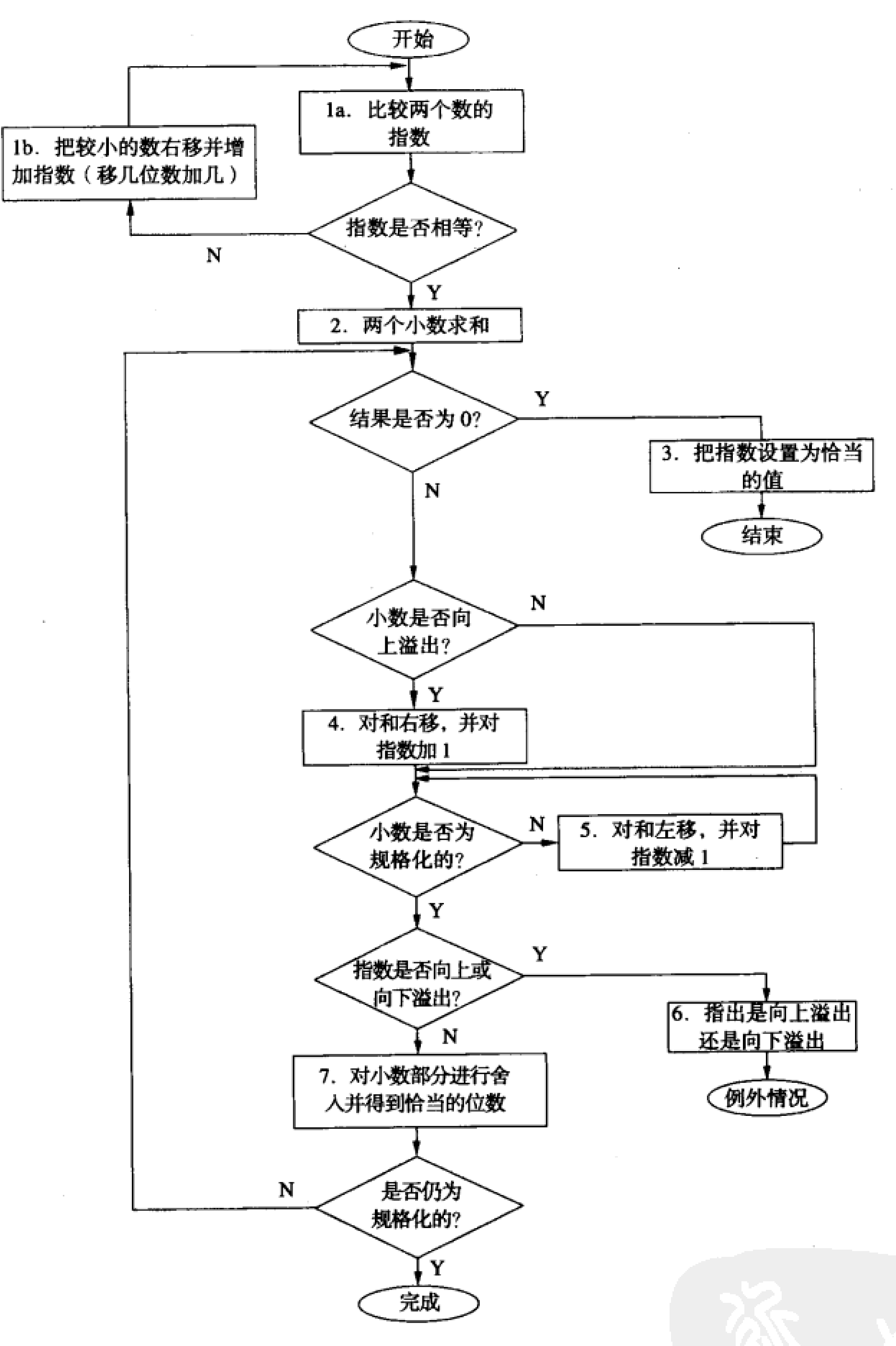


图 7.12　浮点数加法流程图

在进行4位小数相加时，如果$|E_1-E_2|>3$，我们就可以跳过移位这一步。对于IEEE单精度数来说，在小数点之后有23位。因此如果两个指数的差大于23，那么认为较小的那个数为0。通常，如果两个指数的差大于允许的小数位数时，我们就可以认为这两个数的和等于较大的那个数。如果$E_1 \gg E_2$，则直接设定$F=F_1$，$E=E_1$；如果$E_2 \gg E_1$，则直接设定$F=F_2$，$E=E_2$。

通过观察浮点数加法的实现步骤，我们可以看出实现一个浮点数加法器必须需要以下硬件单元：

- 用于比较指数的加法器（或减法器）（步骤 1a）。

- 用于将较小数右移的移位寄存器（步骤 1b）。
- 计算小数加法的 ALU（步骤 2）。
- 双向移位器，加 1/减 1 器（步骤 4，5）。
- 向上溢出检测器（步骤 6）。
- 舍入硬件电路（步骤 7）。

这些元件中有很多可以进行合并。例如，存储小数的寄存器也可以用做移位寄存器实现移位操作。存储指数的寄存器也可以用做带有加 1/减 1 功能的计数器。浮点数加法器的硬件电路方案见图 7.13 所示。其主要组成元件是指数比较器和小数加法器。小数的加法可以使用二进制补码的加法算法完成。假设操作数都通过 I/O 总线进行传输。如果操作数是 IEEE 格式中的符号-绝对值计数格式，则可以把它们转化为二进制补码格式，然后进行相加运算。我们应该根据所需的不同格式对各种特殊情况加以解决。计算得到的和被写回加数寄存器，如图 7.13 所示。

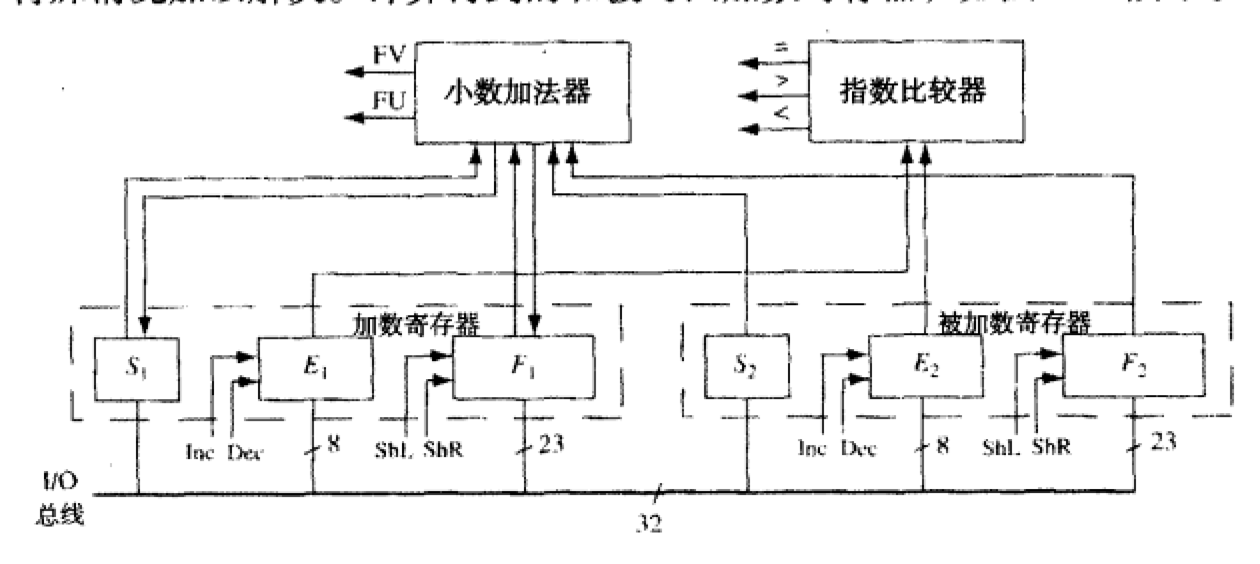


图 7.13　浮点数加法器的简图

图 7.14 给出了浮点数加法器的 VHDL 代码，此加法器是基于 IEEE 单精度浮点数格式的。此代码并没有完全实现这一标准。它解决了等于 0 的特殊情况，但是没有对无穷、非规格化数和 NaN 的情况加以解决。最终结果是采用截尾而不是舍入模式。整个程序中，我们使用了符号-绝对值格式计数法和指数偏置计数法，但是在进行小数加法运算时，我们采用了二进制补码计数法。

FPinput 是一条输入总线，我们假设所有输入数据都是 IEEE 标准格式规格化浮点数。在状态 0，第一个数被载入到 S_1, F_1 和 E_1 中，它们分别表示小数的符号、小数的绝对值和偏置后的指数。当 F_1 被载入时，23 位小数前被加入一个引领位 1，为 0 的特殊情况除外。为 0 时，引领位为 0。同时在 23 位小数末尾加上两个 0，作为保护位和舍入位。在状态 1，要进行相加运算的第二个数被载入到 S_2, F_2 和 E_2 中。在状态 2，选出两个数中指数较小的那个，对小数进行右移，并对指数加 1。当此操作完成时，两个数的指数相同，F_1 或 F_2 为 0 的特殊情况除外。

使用二进制补码算法对两个小数进行加法运算，在程序中是通过并发语句实现的。输入数据首先转化为二进制补码格式，在 F_1 前加上两个符号位(00)。如果 S_1 为 1（为负数），则转化为二进制补码。这里使用了两个符号位，所以当小数加法产生向上溢出时，它只是溢出到第一个符号位，而第二个符号位仍可以表示符号。对 F_2 做类似的处理。以上操作完成后，我们得到两个数——*F1comp* 和 *F2comp*。我们把它们相加并把和赋给 *Addout*。在状态 3 读取加法器输出。*Fsum* 表示小数的绝对值，所以如果 *Addout* 为负数时，则必须对其取补。正常情况下，*Fsum* 的两个符号位应该为“00”，所以可以将这两位舍去，并把结果存回 F_1（F_1 为浮点累加器）。符号位是从 *Addout* 的最高有效位提取的。小数向上溢出和向下溢出分别用 *FV* 和 *FU* 标示。小数向上溢出可以通过对

Addout 的最高两位作异或运算进行检测。此过程也是通过一条并发语句实现的。当小数存在向上溢出时，*Fsum* 的符号位为“01”，所以 *FV*=‘1’，*Fsum* 在存入 F_1 前要向右移一位，并且 E_1 加 1。如果加法运算结果 F_1=0，则在状态 4，E_1 设置为 0，并且浮点数加法运算完成。如果 F_1 是非规格化数，那么在状态 5 对 F_1 进行规格化。此过程可以通过对 F_1 进行左移实现，而且对 F_1 左移几位，就要对 E_1 减几。指数的向上溢出和向下溢出分别用 ovf 和 unf 标示。由于偏置指数的正常取值范围为 1～254，所以当 E_1 减到 0 时，就会出现向下溢出，在状态 5 结束前，应把 unf 置‘1’。在状态 6，如果 E_1=255，则表示指数向上溢出，此时对 ovf 置‘1’。状态 6 结束前发出完成信号。S_1，E_1 和 F_1 通过一条并发语句进行合并，并给出最终结果 *FPsum*。最终结果是 IEEE 格式的。

```
library IEEE;
use IEEE.numeric_bit.all;

entity FPADD is
  port(CLK, St: in bit; done, ovf, unf: out bit;
       FPinput: in unsigned(31 downto 0);  -- IEEE single precision FP format
       FPsum: out unsigned(31 downto 0));  -- IEEE single precision FP format
end FPADD;

architecture FPADDER of FPADD is
 -- F1 and F2 store significand with leading 1 and trailing 0's added
signal F1, F2: unsigned(25 downto 0);
signal E1, E2: unsigned(7 downto 0);  -- exponents
signal S1, S2, FV, FU: bit;
 -- intermediate results for 2's complement addition
signal F1comp, F2comp, Addout, Fsum: unsigned(27 downto 0);
signal State: integer range 0 to 6;
begin  -- convert fractions to 2's comp and add
  F1comp <= not ("00" & F1) + 1 when S1 = '1' else "00" & F1;
  F2comp <= not ("00" & F2) + 1 when S2 = '1' else "00" & F2;
  Addout <= F1comp + F2comp;
   -- find magnitude of sum
  Fsum <= Addout when Addout(27) = '0' else not Addout + 1;
  FV <= Fsum(27) xor Fsum(26);          -- fraction overflow
  FU <= not F1(25);                     -- fraction underflow
  FPsum <= S1 & E1 & F1(24 downto 2); -- pack output word
  process(CLK)
  begin
    if CLK'event and CLK = '1' then
      case State is
        when 0 =>
          if St = '1' then -- load E1 and F1
            E1 <= FPinput(30 downto 23); S1 <= FPinput(31);
            F1(24 downto 0) <= FPinput(22 downto 0) & "00";
            -- insert 1 in significand (or 0 if the input number is 0)
            if FPinput = 0 then F1(25) <= '0'; else F1(25) <= '1'; end if;
            done <= '0'; ovf <= '0'; unf <= '0'; State <= 1;
          end if;
        when 1 =>  -- load E2 and F2
          E2 <= FPinput(30 downto 23); S2 <= FPinput(31);
          F2(24 downto 0) <= FPinput(22 downto 0) & "00";
          if FPinput = 0 then F2(25) <= '0'; else F2(25) <= '1'; end if;
          State <= 2;
        when 2 =>  -- unnormalize fraction with smallest exponent
         if F1 = 0 or F2 = 0 then State <= 3;
         else
           if E1 = E2 then State <= 3;
           elsif E1 < E2 then
             F1 <= '0' & F1(25 downto 1); E1 <= E1 + 1;
           else
             F2 <= '0' & F2(25 downto 1); E2 <= E2 + 1;
           end if;
         end if;
        when 3 => -- add fractions and check for fraction overflow
          S1 <= Addout(27);
          if FV = '0' then F1 <= Fsum(25 downto 0);
          else F1 <= Fsum(26 downto 1); E1 <= E1 + 1; end if;
          State <= 4;
        when 4 => -- check for sum of fractions = 0
          if F1 = 0 then E1 <= "00000000"; State <= 6;
          else State <= 5; end if;
        when 5 => -- normalize
          if E1 = 0 then unf <= '1'; State <= 6;
          elsif FU = '0' then State <= 6;
          else F1 <= F1(24 downto 0) & '0'; E1 <= E1 - 1;
          end if;
        when 6 => -- check for exponent overflow
          if E1 = 255 then ovf <= '1'; end if;
          done <= '1'; State <= 0;
      end case;
    end if;
  end process;
end FPADDER;
```

图 7.14 浮点数加法器的 VHDL 程序

针对下面的情况下对浮点数加法器进行了测试。

加数		被加数		期望结果	
数（二进制）	IEEE 单精度	数（二进制）	IEEE 单精度	数（二进制）	IEEE 单精度
0	x00000000	0	x00000000	0	x00000000
1×2^0	x3F800000	1×2^0	x3F800000	1×2^1	x40000000
-1×2^0	xBF800000	-1×2^0	xBF800000	-1×2^1	xC0000000
1×2^0	x3F800000	-1×2^0	xBF800000	0	x00000000
$1.111...\times2^{127}$	x7F7FFFFF	1×2^0	x3F800000	$1.111...\times2^{127}$	x7F7FFFFF
$-1.111...\times2^{127}$	xFF7FFFFF	-1×2^0	xBF800000	$-1.111...\times2^{127}$	xFF7FFFFF
$1.111...\times2^{127}$	x7F7FFFFF	$1.111...\times2^{127}$	x7F7FFFFF	向上溢出	
$-1.111...\times2^{127}$	xFF7FFFFF	$-1.111...\times2^{127}$	xFF7FFFFF	向上溢出	
1.11×2^8	x43E00000	-1.11×2^6	xC2E00000	1.0101×2^8	x43A80000
-1.11×2^8	xC3E00000	1.11×2^6	x42E00000	-1.0101×2^8	xC3A80000
$1.111...\times2^{127}$	x7F7FFFFF	$0.0...01\times2^{127}$	x73800000	向上溢出	
$-1.111...\times2^{127}$	xFF7FFFFF	$-0.0...01\times2^{127}$	xF3800000	向上溢出	
$1.1...10\times2^{127}$	x7F7FFFFF	$0.0...01\times2^{127}$	x73800000	$1.111...\times2^{127}$	x7F7FFFFF
$-1.1...10\times2^{127}$	xFF7FFFFF	$-0.0...01\times2^{127}$	xF3800000	$-1.111...\times2^{127}$	xFF7FFFFF
1.1×2^{-126}	x00C00000	-1.0×2^{-126}	x80800000	向下溢出	

7.4　浮点数的其他运算

7.4.1　减法

浮点数的减法与加法基本相同，除了把步骤 2 中的小数相加换成相减外，其他步骤完全相同。

7.4.2　除法

两个浮点数的商是

$$(F_1\times2^{E_1})\div(F_2\times2^{E_2})=(F_1/F_2)\times2^{(E_1-E_2)}=F\times2^E$$

因此，浮点数相除的基本法则就是小数相除、指数相减。浮点数除法与浮点数乘法运算类似，也存在一些特殊情况。在做除法之前，我们必须检测一下除数是否是 0。如果 F_1 和 F_2 是规格化的，那么最大的正商(F)为

$$0.111.../0.1000...=01.111...$$

它小于 10_2，所以小数的向上溢出容易纠正。例如，

$$(0.110101\times2^2)\div(0.101\times2^{-3})=01.010\times2^5=0.101\times2^6$$

另外，如果 $F_1\gg F_2$，在做除法之前我们可以对 F_1 右移以避免小数的向上溢出。在 IEEE 格式中，当除数为 0 时，结果设置为 NaN（非数值）。

本章中，我们介绍了浮点数的不同表达形式，对 IEEE 单精度和双精度格式进行了讨论。同时，我们还介绍了如何用二进制补码表示浮点数。随后我们对浮点数乘法器加以讨论，还介绍了浮点数加法的运算步骤。在设计乘法器时，我们使用如下步骤。

1. 提出一个浮点数乘法的运算算法，并把所有的特殊情况都包括在内。
2. 画出系统框图，并且对所需的控制信号加以定义。

3. 为控制状态机构建 SM 图（或状态图），并用独立链接的状态机控制小数乘法器。
4. 写出行为描述方式的 VHDL 代码。
5. 测试此 VHDL 代码，并在较高层面验证乘法器设计的正确性。
6. 使用 CAD 软件对乘法器进行综合，然后使用要求的目标技术（如 ASIC 和 FPGA 等）实现此乘法器。

习题

7.1 (a) 8 位二进制补码浮点数格式（4 位为指数，4 位为小数）可以表示的最大的数为多少？

(b) 8 位二进制补码浮点数格式（4 位为指数，4 位为小数）可以表示的最小的数为多少？

(c) IEEE 单精度规格化浮点数可以表示的最大的数为多少？

(d) IEEE 单精度规格化浮点数可以表示的最小的数为多少？

(e) IEEE 双精度规格化浮点数可以表示的最大的数为多少？

(f) IEEE 双精度规格化浮点数可以表示的最小的数为多少？

7.2 把下列十进制数用 IEEE 单精度格式表示：

(i)25.25, (ii)200.25,(iii)1, (iv)0, (v)1000, (vi)8000, (vii)10^6, (viii)–5.4, (ix)1.0×2^{-140}, (x)1.5×10^9

7.3 把下列十进制数用 IEEE 双精度格式表示：

(i)25.25,(ii)200.25, (iii)1, (iv)0,(v)1000, (vi)8000, (vii)10^6, (viii)–5.4, (ix)1.0×2^{-140}, (x) 1.5×10^9

7.4 用 IEEE 单精度格式表示下列十六进制数：

(i)ABABABAB, (ii)45454545, (iii)FFFFFFFF, (iv)0000000, (v)11111111, (vi)01010101

7.5 用 IEEE 双精度格式表示下列十六进制数：

(i)ABABABAB 00000000, (ii)45454545 00000001, (iii)FFFFFFFF 10001000, (iv)0000000 00000000, (v)11111111 10001000, (vi)01010101 01010101

7.6 (a) 用 IEEE 单精度浮点数格式表示–35.25。

(b) 用 IEEE 单精度浮点数格式表示十六进制数 ABCD0000。

7.7 (a) 用 IEEE 单精度浮点数格式表示 25.625。

(b) 用 IEEE 单精度浮点数格式表示–15.6。

7.8 设计一个可以把一个 8 位有符号整数（负数用二进制补码表示）转化成一个浮点数的数字系统。利用与 7.1.1 节中相似的浮点数格式进行设计，此题中小数为 8 位，指数为 4 位。要求小数是规格化的。

(a) 画出系统框图并为此转化操作设计一个算法。假设整数已经载入到 8 位寄存器中,而且当转化完成时,小数也应该在同一个寄存器中。把–27 转化为浮点数，并以此演示你的算法。

(b) 画出控制器图状态图。假设开始信号只持续一个时钟（两个状态足以满足要求）。

(c) 写出系统的 VHDL 程序。

7.9 (a) 给出下面两个浮点数的积，并把结果进行适当的规格化。假设使用 4 位二进制补码格式。

$$F_1 = 1.011, E_1 = 0101, F_2 = 1.001, E_2 = 0011$$

(b) 如果 $F_1 = 1.011, E_1 = 1011, F_2 = 0.110, E_2 = 1101$，重做(a)。

7.10 一个浮点数用 4 位小数和 4 位指数表示，并且负数用二进制补码形式表示。设计一个高效的系统以实现此数与–4 相乘。要求考虑所有的特殊情况，并给出规格化的结果。假设小数

的初始值是规格化的或为 0。注意：此系统只乘以–4。

(a) 举例说明何时为正常情况，何时为特殊情况（针对乘以–4）。

(b) 画出系统框图。

(c) 画出控制单元的 SM 图，并对所有使用的符号加以定义。

7.11 重新设计图 7.7 中的浮点数乘法器，用一个普通的与总线相连接的 5 位全加器代替小数和指数的两个独立的加法器。

(a) 重新画出系统框图，确保包括所有与总线的链接和所有的控制信号。

(b) 画出新控制器的新 SM 图。

(c) 写出此乘法器的 VHDL 程序，或者明确说明已有的程序中需要改动之处。

7.12 设计一个可以计算浮点数平方的电路。浮点数为 $F\times2^E$，其中 F 是规格化的 5 位小数，E 是 5 位整数。负数用二进制补码表示，计算结果也应该是规格化的。本题中可以使用性质 $(-F)^2 = F^2$。

(a) 画出电路的框图（只使用一个加法器和一个求补器）。

(b) 阐述你的计算步骤，要求考虑所有特例，并用 F=1.0110, E=00100 对你的计算步骤进行说明。

(c) 画出主控制器 SM 图。你可以假设乘法操作是通过一个单独的控制电路来实现的，且当乘法完成时输出 $Mdone = 1$。

(d) 写出系统的 VHDL 程序。

7.13 写出 IEEE 单精度浮点数格式下，浮点数乘法器的行为描述方式 VHDL 代码。要求使用重载乘法操作符代替相加-移位乘法器。忽略无穷、非规格化数和 NaN 的情况，并且对最终结果进行截尾，替代舍入。

7.14 写出图 7.14 中浮点数加法器的测试平台。

7.15 把下面两个浮点数相加（给出每一步的计算过程）。假设每个小数都是 5 位的（包括符号位），每一个指数也是 5 位的（包括符号位），且负数用二进制补码表示。

$$F_1 = 0.1011 \qquad E_1 = 11111$$
$$F_2 = 1.0100 \qquad E_2 = 11101$$

7.16 求两个浮点数的和，生成的和也是浮点数：

$$(F_1\times2^{E_1})+(F_2\times2^{E_2}) = F\times2^E$$

假设 F_1, F_2 都是规格化的，并且最后的结果也是规格化的。

(a) 列出执行浮点数加法的必要步骤，包括所有的特例。

(b) 用 $F_1 = 1.0101, E_1 = 1001, F_2 = 0.1010, E_2 = 1000$ 来解释这些步骤。注意，小数是 5 位的（包括符号位），指数是 4 位的（包括符号位）。

(c) 写出系统的 VHDL 程序。

7.17 改变图 7.14 中浮点数加法器的 VHDL 代码，使代码可以满足以下条件：

(a) 允许输入和输出为 IEEE 标准单精度非规格化数。

(b) 在状态 2，当两指数的差大于 23 时可以进行快速处理。

(c) 对于结果中的小数部分采用向上舍入，而不是截尾。

7.18 (a) 求两个浮点数的和：$0.111\times2^5 + 0.101\times2^3$，并对结果进行规格化。

(b) 画出计算($F_1 \times 2^{E_1}$)与 $F_1 \times 2^{E_2}$)和的浮点数加法器的 SM 图。假设小数部分初始化为规格化（或 0）。当小数部分为 0 时，指数应为–8。如果最终结果的指数部分产生向上溢出，则把指数溢出标志位(EV)置 1。每个加数的小数部分为 4 位，指数部分也为 4 位，负数均用二进制补码形式表示。假设当接收到开始信号(St)时，所有寄存器(F_1,E_1,F_2,E_2)在一个时钟内载入数据。如果 $E_1 > E_2$，则控制信号 GT=1；如果 $E_1 < E_2$，则控制信号 LT=1。要求对所有使用的控制信号加以定义。要求包括 $|E_1 - E_2| > 3$ 的特殊情况。

7.19 (a) 画出浮点数减法器的框图。假设减法器输入端的数据都是规格化的，并且结果也是规格化的。小数和指数都是 8 位的（包括符号位），且负数均用二进制补码表示。

(b) 画出浮点数减法器控制电路的 SM 图。要求对使用的控制信号均加以定义。对于用做控制电路输入的每个控制信号均给出一个表达式。

(c) 写出浮点数减法器的 VHDL 程序。

7.20 (a) 写出进行浮点数减法的必要步骤，包括特殊情况。假设初始数值都是规格化的，并且最后结果也是规格化的。

(b) 对下面的两个数做减法（小数是二进制补码形式）：

$$(1.0111 \times 2^{-3}) - (1.0101 \times 2^{-5})$$

(c) 写出系统的 VHDL 程序。小数为 5 位（包括符号位），指数为 4 位（包括符号位）。

7.21 设计一个浮点数除法器:

$$(F_1 \times 2^{E_1})/(F_2 \times 2^{E_2}) = F \times 2^E$$

假设F_1和F_2是规格化的（或0），负小数均使用二进制补码表示。指数为整数，且负指数均用二进制补码表示。结果若不为0，则也应该是规格化的。小数部分为8位（包括符号位），指数为5位（包括符号位）。

(a) 画出此浮点数除法器的流程图。假设此除法器可以计算两个二进制小数相除，并且结果也是小数。对于小数的除法，在流程图中不要显示详细步骤，只标出“除以”即可。在进行除法 操作前，要求 $|F_2| > |F_1|$。

(b) 通过计算

$$0.111 \times 2^3 / 1.011 \times 2^{-2}$$

来解释除法的运算步骤。在计算 F_1 除以 F_2 时，不必显示详细步骤，只需显示最后的结果。

(c) 写出此系统的 VHDL 程序。

7.22 假设 A, B, C 均为浮点数，均按照 IEEE 单精度浮点数格式表示。下面进行浮点数加法运算。如果 $A = 2^{40}, B = -2^{40}, C = 1$，则

$A+(B+C)$ 为多少？（先计算 $B+C$，再把结果与 A 相加）

$(A+B)+C$ 为多少？（先计算 $A+B$，再把结果与 C 相加）

7.23 假设 A, B, C 均为浮点数，均按照 IEEE 双精度浮点数格式表示。下面进行浮点数加法运算。

如果 $A = 2^{40}, B = -2^{40}, C = 1$，则

$A+(B+C)$ 为多少？（先计算 $B+C$，再把结果与 A 相加）

$(A+B)+C$ 为多少？（先计算 $A+B$，再把结果与 C 相加）

7.24 假设 A, B, C 均为浮点数，均按照 IEEE 单精度浮点数格式表示。下面进行浮点数加法运算。

如果 $A = 2^{65}, B = -2^{65}, C = 1$，则

$A+(B+C)$ 为多少？（先计算 $B+C$，再把结果与 A 相加）

$(A+B)+C$ 为多少？（先计算 $A+B$，再把结果与 C 相加）

7.25 假设 A, B, C 均为浮点数，均按照 IEEE 双精度浮点数格式表示。下面进行浮点数加法运算。

如果 $A = 2^{65}, B = -2^{65}, C = 1$，则

$A+(B+C)$ 为多少？（先计算 $B+C$，再把结果与 A 相加）

$(A+B)+C$ 为多少？（先计算 $A+B$，再把结果与 C 相加）



第 8 章　VHDL 语言的高级议题

在前面的章节中，我们已经介绍了 VHDL 的基本特征及如何在数字系统设计中使用 VHDL。在本章中，我们将通过介绍 VDHL 的其他特性具体说明 VHDL 的强大功能和灵活性。我们介绍了 VHDL 函数语句和过程语句，还将介绍很多其他的特性，如属性语句、函数重载语句、类属语句和生成语句。此外，我们对 IEEE 多值逻辑系统和信号分辨原理也加以介绍。我们还给出了一个简单的存储模型以说明三态信号的使用。

8.1　函数语句

VLSI 电路的一个主要特点就是重复使用类似的结构。VHDL 提供了函数语句和过程语句，使用它们可以很容易地引用具有相同功能的模块，或者重复使用相同的结构。本节中，我们将对函数语句加以介绍。通过返回语句，函数只能返回一个值，而与函数相比，过程要通用和复杂得多，它可以通过使用输出变量返回任意多的值。我们将在下一节中介绍过程语句。

一个函数执行一个顺序算法，把一个值返回给主程序。下面例子中，当函数被调用时，它把输入位矢量(*reg*)循环右移一位，然后将其返回给主调程序：

```
function rotate_right (reg: bit_vector)
return bit_vector is
begin
return reg ror 1;
end rotate_right;
```

只要可以使用表达式的地方，就可以使用函数调用，例如，设 A = "10010101"，则语句

```
B < = rotate_right(A);
```

的结果为 B = "11001010"，而且 A 的值不变。

函数说明语句的一般格式为

```
function 函数名(形参变量1, 形参变量2, …)
return 返回数据类型  is
      [定义语句]
begin
  顺序语句        -- 必须含有返回值;
end  函数名;
```

函数调用语句的一般格式为

```
函数名(实参变量)
```

实参变量的个数和数据类型必须与函数说明中形参变量的个数和数据类型相匹配。实参变量只作为函数的输入值，在函数执行过程中不发生改变。

例　用VHDL编写一个函数。该函数可以生成一个4位数生成一个偶校验位。输入为一个4位数，输出为一个码字，包括数据位和校验位。结果如图8.1所示。

```
-- Function example code without a loop
-- This function takes a 4-bit vector
-- It returns a 5-bit code with even parity

function parity (A: bit_vector(3 downto 0))
  return bit_vector is

variable parity: bit;
variable B: bit_vector(4 downto 0);
begin
  parity := a(0) xor a(1) xor a(2) xor a(3);
  B := A & parity;
  return B;
end parity;
```

图 8.1　校验位生成函数

如果系统中的几个部分都需要使用奇偶校验电路，那么每次当我们需要使用该电路时就可以调用上面的函数。

图 8.2 所示函数中用到了 **for** 循环语句。在图 8.2 中，循环变量(i)在循环开始时值为 0。随后进入 for 循环语句，执行循环中的顺序语句，并且在 $i = 1, i = 2, i = 3$ 时重复执行这些顺序语句；最后循环结束。

```
-- This function adds two 4-bit vectors and a carry.
-- Illustrates function creation and use of loop
-- It returns a 5-bit sum

function add4 (A, B: bit_vector(3 downto 0); carry: bit)
  return bit_vector is

variable cout: bit;
variable cin: bit := carry;
variable sum: bit_vector(4 downto 0) := "00000";
begin
loop1: for i in 0 to 3 loop
  cout := (A(i) and B(i)) or (A(i) and cin) or (B(i) and cin);
  sum(i) := A(i) xor B(i) xor cin;
  cin := cout;
end loop loop1;
sum(4) := cout;
return sum;
end add4;
```

图 8.2　求和函数

当 A, B 和 C 为整数时，执行语句 C<=A+B 后，C 等于 A 与 B 的和，但是如果 A, B 和 C 均为位向量时，上面的语句将不能执行，这是因为“ + ”操作符不能用于位向量求和。但是我们可以编写一个计算位向量加法的函数，如图 8.2 中的函数可以对两个 4 位向量及一个进位求和，并返回一个 5 位的向量。其函数名为 *add4*；形参变量为 *A*, *B* 和 *carry*；返回值为位向量。变量 *cout* 和 *cin* 用于记录计算的中间值。变量 *sum* 用于存储返回值。当该函数被调用时，*cin* 的初始值等于 *carry* 的值，我们使用 **for** 循环并按照与串行加法器相同的方法把 *A*, B 的值按位串行相加。当循环第一次执行时，*sum*(0)由 *A*(0), *B*(0)及 *cin* 的初始值计算得到，接着把新得到的 *cout* 的值赋给 *cin*；然后循环反复执行。当循环第二次执行时，*sum*(1)由 *A*(1), *B*(1)及 *cin* 的新值计算得到。当循环执行 4 次后，所有 *sum*(i)的值都计算完毕，并把 *sum* 返回给主调程序。由于此程序中所有的计算都是由变量进行的，而变量的更新是在瞬间完成的，所以执行函数 *add4* 所需的全部仿真时间为 0。

函数调用语句的格式为

```
add4(A, B, carry)
```

其中 *A*, *B* 和 *carry* 可以用数据类型为位向量且维数为 3 downto 0 的任意表达式替换。例如，语句

```
Z <= add4(X, not Y, '1');
```

也可以调用函数 *add4*。调用函数 *add4* 时，参数 *A*, *B* 和 *carry* 的值分别等于 *X*, not *Y* 和'1'。*X* 和 *Y* 的数据类型必须是维数为 3 **downto** 0 的位向量。所以函数计算

```
Sum = A + B + carry = X + not Y + '1'
```

并且把 *sum* 的值返回给主调程序。因为 *sum* 是变量，所以计算 *sum* 无需附加延时。在 Δ 时间后，*Z* 就等于返回值 *sum*；由于 **not** *Y* + '1'其实就是求 *Y* 的补码，所以求一个数的补码其实就是对其反码加 1。如果忽略 *Z*(4)中存储的进位，则结果为 *Z*(3 **downto** 0) = *X*–*Y*。

函数也可以返回一个数组。下面我们编写这样一个函数，其输入为一组数，返回值也是一组数，并且返回的每个数为输入中对应数的平方。图 8.3 给出了函数和函数调用语句。输入数据的个数也作为一个参数提供给函数。从函数调用语句中可以看出，输入数据的宽度为 4 位。

```
library IEEE;
use IEEE.numeric_bit.all;

entity test_squares is
  port(CLK: in bit);
end test_squares;

architecture test of test_squares is
type FourBitNumbers is array (0 to 4) of unsigned (3 downto 0);
type squareNumbers is array (0 to 4) of unsigned (7 downto 0);
constant FN: FourBitNumbers := ("0001", "1000", "0011", "0010", "0101");
signal answer: squareNumbers;
signal length: integer := 4;

function squares (Number_arr: FourBitNumbers; length: positive)
  return squareNumbers is

variable SN: squareNumbers;
begin
   loop1: for i in 0 to length loop
          SN(i) := Number_arr(i) * Number_arr(i);
   end loop loop1;
return SN;
end squares;

begin
  process(CLK)
  begin
    if CLK = '1' and CLK'EVENT then
     answer <= squares(FN, length);
    end if;
  end process;
end test;
```

图 8.3　计算无符号数数组平方的函数及函数调用

函数广泛地应用于数据类型转换中。我们已经接触了 IEEE numeric_bit 库中的两个数据类型转换函数：`to_integer(A)` 和 `to_unsigned(B, N)`。第一个函数是把一个无符号向量转化为一个整数；第二个函数是把一个整数转化为具有特定位数的无符号向量。

8.2　过程语句

过程语句可以把 VHDL 代码划分为几个模块。函数语句只能返回一个值，与函数语句不同，

过程语句可以通过输出参数返回任意多的值。过程说明语句的格式为

```
procedure  过程名（形参变量列表）  is
           [定义语句]
begin
           顺序语句
end  过程名;
```

形参变量列表规定了过程语句的输入、输出及其数据类型。过程调用的格式为

```
过程名（实参变量列表）;
```

下面，我们编写一个过程 *Addvec*。此过程可以把含有进位的两个 *N* 位向量相加，并返回一个 *N* 位的和向量与一个进位。我们使用的过程调用语句的格式为

```
Addvec(A, B, Cin, Sum, Cout, N);
```

其中 *A*, *B*, *Sum* 是 *N* 位向量；*Cin* 和 *Cout* 的数据类型为位；*N* 为整数。

图 8.4 给出了过程定义。*Add1*, *Add2*, *Cin* 为输入参数；*Sum*, *Cout* 为输出参数；*N* 表示位向量的位数，为正整数。此过程中加法运算算法与函数 *add4* 完全相同。由于在循环过程中每次均需要一个新的 *C* 值，所以 *C* 的数据类型必须为变量（variable）；但是由于 *sum* 没有在循环中出现，所以 *sum* 的数据类型为信号（signal）。当循环 *N* 次后，*sum* 的所有值都计算完毕，但是 *sum* 的值在循环结束后 Δ 时间才得以更新。

在过程说明中每个形参变量的对象类型、模式、数据类型都必须在形变参量列表中加以说明。参变量的对象类型可以是信号(signal)、变量(variable)或常数(constant)。如果输入参变量的对象类型没有说明，那么默认为常数。如果形参变量的对象类型为信号，那么过程调用中的实参变量的对象类型必须是信号；如果形参变量的对象类型为变量，那么过程调用中的实参变量的对象类型必须是变量，但是如果形参变量的对象类型为常数，那么过程调用中的实参变量可以是结果为常数的任意表达式。而且，这个在过程中使用的常数值是不可以更改的，所以类型为常数的形参变量的模式总是 **in 模式**。信号和变量的通信模式可以为 **in**, **out** 或 **inout**。模式为 **out** 或 **inout** 的参变量的值可以在过程中变化，所以用它们返回计算值。

```
-- This procedure adds two n-bit bit_vectors and a carry and
-- returns an n-bit sum and a carry. Add1 and Add2 are assumed
-- to be of the same length and dimensioned n-1 downto 0.

procedure Addvec (Add1, Add2: in bit_vector; Cin: in bit;
                  signal Sum: out bit_vector; signal Cout: out bit;
                  n: in positive) is

variable C: bit;
begin
  C := Cin;
  for i in 0 to n-1 loop
    Sum(i) <= Add1(i) xor Add2(i) xor C;
    C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
  end loop;
  Cout <= C;
end Addvec;
```

图 8.4　对位向量求和的过程

在过程 *Addvec* 中，参数 *Add1*, *Add2* 和 *Cin* 均选用默认对象类型——常数。因此，在过程调用语句中 *Add1*, *Add2* 和 *Cin* 可以用结果为常数的任意表达式代替。由于 *Sum*, *Cout* 的值在过程中有所改变而且用做返回量，所以它们的对象类型为信号(signal)。因此，在过程调用语句中 *Sum*, *Cout* 只能用适当的类型和维数的信号代替。

函数说明语句中的形参变量列表与过程说明语句相似，但是在函数语句中变量（**variable**）类型形参变量不允许使用。由于一个函数只返回一个值，并且该值不能用一个参变量返回，所以函数语句中的所有参变量的模式都必须是 **in**（默认），不允许为 **out** 和 **inout**。表 8.1 总结了在函数语句和过程语句中所有可以使用的参变量模式和对象类型。过程输出参变量的模式可以是 **out** 或 **inout**，其对象类型可以是信号或变量。显然，过程输出参变量的对象类型不可以是常数，因为常数的值不能被更改。

表 8.1 子程序调用的参数

模式	类型	实参变量	
		过程调用	函数调用
In[1]	常数[2]	表达式	表达式
	信号	信号	信号
	变量	变量	n/a
Out/inout	信号	信号	n/a
	变量[3]	变量	n/a

1. 函数的默认模式。
2. 模式 in 的默认类型。
3. 模式 out/inout 的默认类型。
   注意：n/a 指不可用。

8.3 属性语句

属性（attributes）是 VHDL 语言的一个重要特性，它既可以同信号联系在一起，也可以同数组联系在一起。

8.3.1 信号属性语句

在之前的章节中我们已经使用了一个信号属性——'EVENT 属性，用来创建边沿触发的时钟。我们知道，在信号 *CLOCK* 发生改变的同时 CLOCK'EVENT 为真。VHDL 语言中有两种属性语句：(1)返回一个值的属性语句；(2)返回一个信号的属性语句。

表 8.2 给出了几个返回一个值的属性语句。表中，*S* 代表一个信号名，并且 *S* 和属性的名称用一个单引号来分隔开。在 VHDL 中，在信号上加上一个事件表示这个信号发生了变化。这样，如果 *S* 发生事务，则 S'ACTIVE（读做："S tick ACTIVE"）将返回一个真值（TRUE）。不论信号本身变化与否，每次只要有事务发生，信号就会被重新赋值。考虑下面的并发 VHDL 语句：`A<=B and C`。如果 $B=0$，那么每次 *C* 变化时，*A* 都将变化，因为每次 *C* 变化时 *A* 都被重新计算。如果 $B=1$，那么每次 *C* 变化时 *A* 都会发生一个事件和一个事务。如果 *S* 被重新赋值，则即使 *S* 没有变化，S'ACTIVE 也将返回一个真值。相对地，只有当 *S* 发生变化时，S'EVENT 才返回 TRUE。如果在 *T* 时刻 *S* 的值发生变化，那么 S'EVENT 的值在 *T* 时刻为真，而在 $T+\Delta$ 时刻为假。

表 8.2 返回单一值的信号属性

属性	返回值
S'EVENT	若信号在当前 Δ 时间内发生事件，则返回真，否则返回假
S'ACTIVE	若信号在当前 Δ 时间内发生事务，则返回真，否则返回假
S'LAST_EVENT	*S* 的从上一次事件发生到目前的时间差
S'LAST_VALUE	上一次事件发生之前的 S 值
S'LAST_ACTIVE	*S* 从上一次事务发生到目前的时间差

表 8.3 给出了生成信号的信号属性。(time) 外面的方括号表明 (time) 是可选的。如果 (time) 被省略，那么使用 Δ 时间。属性 S 'DELAYED (time) 生成一个与 *S* 同类型的信号，但是它要延迟指定的时间。图 8.5 中的例子说明了表 8.3 中列出的属性的用法。信号 *C_delayed5* 为信号 *C* 右移 5 ns。信号 *A_trans* 在每次 *B* 或者 *C* 变化时跳变，因为不论何时 *B* 或 *C* 变化，都会给 *A* 产生一个事务 (transaction)。语句 A **<=** B **and** C 刚开始计算时，在 Δ 时刻为 *A* 产生一个事务，所以 *A_trans* 在此时变成'1'。如果 *A* 在上述的时间间隔内没有发生变化，那么信号 A 'STABLE(time) 为真。这样，*A_stable5* 在 *A* 变化后 5 ns 内为假，否则为真。如果 *A* 在上述时间间隔之内没有发生事务，那么信号 A 'QUIET (time) 为真。这样，*A'quiet5* 在 *A* 发生事务后的 5 ns 为假。如果在当前 Δ 时间内有事件发生，那么 S'EVENT 和 **not** S'STABLE 都返回 '真'；但是它们不能替换使用，因为前者返回一个值而后者返回一个信号。

表 8.3　生成信号的信号属性

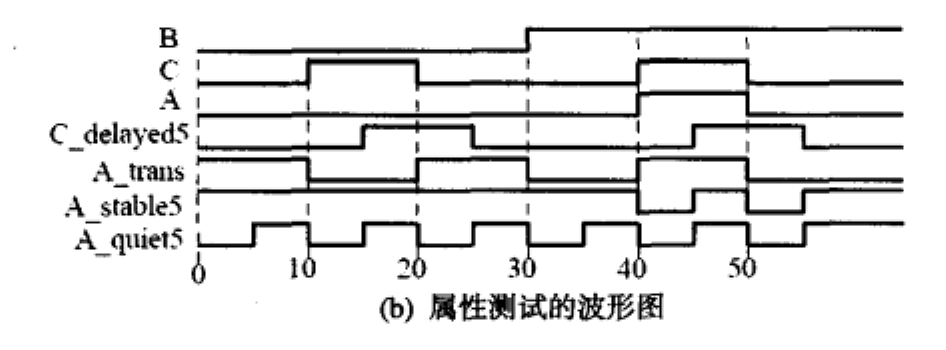
属　性	生成的信号
S 'DELAYED[(time)]*	与 *S* 同类型的信号，但是比信号 S 延迟指定的时间
S 'STABLE[(time)]*	布尔信号（若 *S* 在指定时间内无事件发生，则布尔信号为真）
S 'QUIET[(time)]*	布尔信号（若 *S* 在指定时间内无事务发生，则布尔信号为真）
S 'TRANSACTION	bit 类型的信号。每当 *S* 发生事务时，此信号就发生变化

*当没有指定时间时，时间为 Δ 。

```
entity attr_ex is
   port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
   signal A, C_delayed5, A_trans : bit;
   signal A_stable5, A_quiet5 : boolean;
begin
   A <= B and C;
   C_delayed5 <= C'delayed(5 ns);
   A_trans <= A'transaction;
   A_stable5 <= A'stable(5 ns);
   A_quiet5 <= A'quiet(5 ns);
end test;
```

(a) 属性测试的 VHDL 代码



(b) 属性测试的波形图

图 8.5　信号属性示例

8.3.2　数组属性语句

表 8.4 给出了数组属性语句。表中，*A* 既可以作为数组名称也可以作为数组类型。表中所举的例子中，*ROM1* 是一个二维数组，其第一个下标的取值范围是 0 to 15，第二个下标的取值范围是 7 downto 0。由于第二个下标的左边界为 7，所以 ROM1'LEFT(2)为 7。尽管 *ROM1* 是作为信号进行说明的，但是数组属性对数组常数和数组变量同样有效。在表中所举的例子中，如果把 *ROM1* 用它的类型 ROM 替换，那么结果是相同的。对于一个向量（一维数组）来说，如果 *N* 为 1，则它可以被省略。如果 *A* 是下标取值范围为 2 to 9 的位向量，则 A'LEFT 为 2，A'LENGTH 为 8。

表 8.4　数组属性

```
type ROM is array (0 to 15, 7 downto 0) of bit;
signal ROM1 : ROM;
```

属性	返回	示例
A'LEFT(N)	第 N 个下标取值范围的左边界	ROM1'LEFT(1) = 0 ROM1'LEFT(2) = 7
A'RIGHT(N)	第 N 个下标取值范围的右边界	ROM1'RIGHT(1) = 15 ROM1'RIGHT(2) = 0
A'HIGH(N)	第 N 个下标取值范围的最大边界	ROM1'HIGH(1) = 15 ROM1'HIGH(2) = 7
A'LOW(N)	第 N 个下标取值范围的最小边界	ROM1'LOW(1) = 0 ROM1'LOW(2) = 0
A'RANGE(N)	第 N 个下标取值范围	ROM1'RANGE(1) = 0 **to** 15 ROM1'RANGE(2) = 7 **downto** 0
A'REVERSE_RANGE(N)	第 N 个下标取值范围的反序表示	ROM1'REVERSE_RANGE(1) = 15 **downto** 0 ROM1'REVERSE_RANGE(2) = 0 **to** 7
A'LENGTH(N)	第 N 个下标取值范围的大小	ROM1'LENGTH(1) = 16 ROM1'LENGTH(2) = 8

8.3.3　属性语句的使用

在错误检测中属性语句经常与 assert 语句联合使用（见 2.19 节）。assert 语句用于检查某个条件是否为真，如果不是，则显示一个错误信息。下面我们举两个例子，一个用于说明如何使用信号属性语句，另一个说明如何使用数组属性语句。

信号属性的使用

观察图 8.6 所示进程。此进程检查启动时间和保持时间是否满足 D 触发器的要求。我们使用属性'EVENT 和'STABLE。如果信号在指定的时间内没有事件发生，则'STABLE 返回一个布尔信号（例如，若返回"真"，则表示信号在给定的时间内稳定）。例如，若 *A* 在给定的时间间隔(time)内没有发生变化，则信号 A'STABLE(time)为真。这样，A'STABLE(5)会在 *A* 改变后 5 ns 为假，否则为真。

```
check: process
begin
  wait until (Clk'event and CLK = '1');
  assert (D'stable(setup_time))
    report ("Setup time violation")
    severity error;
  wait for hold_time;
  assert (D'stable(hold_time))
    report ("Hold time violation")
    severity error;
end process check;
```

图 8.6　建立时间和保持时间的测试进程

在 check 进程中，当时钟有效沿到来时，输入 *D* 被检测以验证其在指定的 *setup_time* 时间内是否稳定。如果不是，则报告一个建立时间违规错误。然后对保持时间进行检测，验证 *D* 是否在指定的 *hold_time* 时间内稳定，如果不是，则报告一个保持时间违规错误。

向量加法中数组属性的使用

图 8.7 所示的计算位向量加法的过程中联合使用了 assert 语句和数组属性语句。此过程可以求两个任意长度位向量的和。但是，在计算时向量的长度必须相同。由于在过程调用时没有传递

数组的长度信息，所以向量的长度没有作为一个变量被传递到过程中，因此此过程中使用数组属性语句来检测向量的长度是否相等。过程 *Addvec2* 的 VHDL 代码见图 8.7。此过程的输入为两个输入向量和一个数据类型为位的进位。当过程 *Addvec2* 被执行时，它生成一个临时变量 *C* 作为内部的进位，并且将它初始化为输入进位 *Cin*。然后产生别名 *n*1，*n*2 和 *S*，它们分别与 *Add*1，*Add*2 和 *Sum* 的长度相同。这些别名的维数的取值范围为–1 downto 0。虽然 *Add1*, *Add2* 和 *Sum* 的取值范围可以是 **downto** 或 **to**，但是不包含 0。为了使以后的运算简单化，这些别名都用同一种方式进行定义。如果输入的向量和 *Sum* 的长度不同，那么将产生一个错误报告信息。在循环中，和与进位都是一位一位计算的。由于循环必须从 *i* = 0 开始，所以 *i* 的范围与 *S* 相反。最后，把相应的临时变量 *C* 的值赋给进位输出 *Cout*。

```
-- This procedure adds two bit_vectors and a carry and returns a sum
-- and a carry. Both bit_vectors should be of the same length.

procedure Addvec2 (Add1, Add2: in bit_vector; Cin: in bit;
                   signal Sum: out bit_vector;
                   signal Cout: out bit) is

variable C: bit := Cin;
alias n1: bit_vector(Add1'length-1 downto 0) is Add1;
alias n2: bit_vector(Add2'length-1 downto 0) is Add2;
alias S: bit_vector(Sum'length-1 downto 0) is Sum;
begin
  assert ((n1'length = n2'length) and (n1'length = S'length))
    report "Vector lengths must be equal!"
    severity error;
  for i in s'reverse_range loop   -- reverse range makes you start from LSB
    S(i) <= n1(i) xor n2(i) xor C;
    C := (n1(i) and n2(i)) or (n1(i) and C) or (n2(i) and C);
  end loop;
  Cout <= C;
end Addvec2;
```

图 8.7 计算位向量加法的过程

8.4 重载操作符的生成

下面我们介绍如何构建重载操作符。重载操作符是指把某种操作符从已经定义的默认数据类型扩展到其他数据类型。此操作符将隐式调用一个恰当的函数，这样就不用显示地调用函数或过程了。当编译器遇到一个函数，且其函数名为带双引号的操作符时，编译器就把此函数作为重载操作符函数处理。

VHDL 的算术操作符“ + ”和“–”是用于整数操作的，对位向量操作没有定义。为了对用于无符号位向量的重载算术操作符进行访问，我们使用了 IEEE numeric_bit 库。下面我们构建一个用于位向量的“ + ”函数。

图 8.8 中的包集合展示了如何创建一个用于位向量的“+”函数。此函数把两个位向量相加并返回一个位向量。此函数使用了别名，这样它与位向量的取值范围是相互独立的，但是它假定向量的长度均相同。在程序中，我们使用一个 for 循环对每一位进行加法运算。如果没有此重载函数，则“ + ”函数将不能应用于位向量，并且 IEEE numeric_bit 库也只支持无符号类型的加法运算。

重载也可以应用于过程和函数中。几个过程可以拥有相同的名字，并且过程调用中实参的数据类型决定了进行何种过程调用。在 IEEE numeric_bit 库中对多个重载操作符和重载函数都进行了定义。

```
-- This package provides an overloaded function for the plus operator

package bit_overload is
  function "+" (Add1, Add2: bit_vector)
    return bit_vector;
end bit_overload;

package body bit_overload is
  -- This function returns a bit_vector sum of two bit_vector operands
  -- The add is performed bit by bit with an internal carry
  function "+" (Add1, Add2: bit_vector)
    return bit_vector is
  variable sum: bit_vector(Add1'length-1 downto 0);
  variable c: bit := '0';                   -- no carry in
  alias n1: bit_vector(Add1'length-1 downto 0) is Add1;
  alias n2: bit_vector(Add2'length-1 downto 0) is Add2;
  begin
    for i in sum'reverse_range loop
      sum(i) := n1(i) xor n2(i) xor c;
      c := (n1(i) and n2(i)) or (n1(i) and c) or (n2(i) and c);
    end loop;
    return (sum);
  end "+";
end bit_overload;
```

图 8.8　带有用于位向量的过载操作符的 VHDL 包集合

8.5　多值逻辑和信号分辨

在前面章节的 VHDL 代码中，我们已经使用了双值逻辑。为了表示三态门和总线，我们需要引入第三个值'Z'，用它来代表高阻状态。另外有时还需要引入第四个值'X'，用来代表未知状态。如果信号的初值未知或者一个信号同时被赋予两个相互冲突的值（例如'0'和'1'），那么将会出现未知状态。如果一个门的输入是'Z'，那么此门的输出可以赋予一个未知值'X'。

为了满足这些需求，我们必须使用多值逻辑。IEEE numeric_std 和 IEEE 标准逻辑均使用 9 值逻辑。不同的 CAD 工具开发商定义了 7 值、9 值和 11 值的逻辑之间的转换。

本章中，我们将介绍两个多值逻辑的例子：(1) 一个 4 值逻辑系统；(2) IEEE-1164 标准 9 值逻辑系统。我们将在 8.5.1 节中介绍 4 值逻辑系统，在 8.6 节中介绍 9 值逻辑系统。

8.5.1　4 值逻辑系统

4 值逻辑中的信号可以被赋值为'X', '0', '1'和'Z'。这 4 个符号的含义如下所示：

'X'	未知
'0'	0
'1'	1
'Z'	高阻

高阻状态用于模拟三态缓冲器和总线。如果信号的初值未知或者一个信号同时被赋予两个相互冲突的值（例如'0'和'1'），那么可以使用未知状态。

下面我们用 4 值逻辑模拟一个三态缓冲器。图 8.9 是输出被接到一起的两个三态缓冲器，图 8.10 是它的 VHDL 描述。这个例子中使用了一个新的数据类型 X01Z，它可以被赋值为'X', '0', '1'和'Z'。三态缓冲器有一个高电平有效的使能端，所以当 $b = 1$ 且 $d = 0$ 时，$f = a$；当 $b = 0$ 且 $d = 1$ 时，$f = c$；当 $b = d = 0$ 时，输出 f 为高阻状态 Z。如果 $b = d = 1$，则产生输出冲突。程序

中，我们列出了两个构造体，第一个用了两个并发语句，而第二个用了两个进程。在每种情况下，f 均被两个不同的源所驱动，VHDL 采用了一个分辨函数（resolution function）来决定实际的输出是什么。例如，如果 $a = c = d = 1$ 且 $b = 0$，则在一种情况下 f 被一个并发语句或者进程设为'Z'，在另一种情况下 f 被另一个并发语句或者进程设为'1'。系统自动调用分辨函数来决定 f 的正确值为'1'。如果 f 同时被赋予'0'和'1'，那么分辨函数将 f 视为未知值'X'。

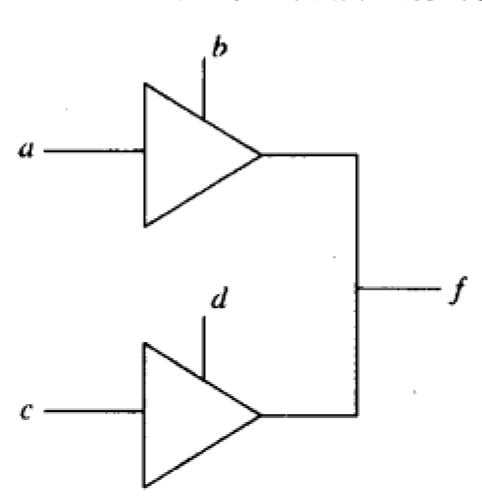


图 8.9　带有高电平使能端的三态缓冲器

```
use WORK.fourpack.all; -- fourpack is a resolved package for 4-variable logic
                       -- more details on resolution in next subsection

entity t_buff_exmpl is
  port(a, b, c, d: in X01Z;  -- signals are four-valued
       f: out X01Z);
end t_buff_exmpl;

architecture t_buff_conc of t_buff_exmpl is
begin
  f <= a when b = '1' else 'Z';
  f <= c when d = '1' else 'Z';
end t_buff_conc;
architecture t_buff_bhv of t_buff_exmpl is
begin
  buff1: process(a, b)
  begin
    if (b = '1') then
      f <= a;
    else
      f <= 'Z';  -- "drive" the output high Z when not enabled
    end if;
  end process buff1;

  buff2: process(c, d)
  begin
    if (d = '1') then
      f <= c;
    else
      f <= 'Z';  -- "drive" the output high Z when not enabled
    end if;
  end process buff2;
end t_buff_bhv;
```

图 8.10　三态缓冲器的 VHDL 代码

图 8.10 的代码中使用了 4 值逻辑包集合和相应的信号分辨函数。下面我们介绍一下如何创建信号分辨函数。如果想使图 8.10 中的代码正常工作，则必须使用下面一节中的包集合。

8.5.2　信号分辨函数

VHDL 中的信号可以是已分辨信号或未分辨信号。当一个系统中用不同的线驱动同一个

信号路径时，信号分辨就十分必要了。当两个或两个以上的信号都连到同一个点时，信号分辨选择一个值作为最终结果。当多个信号汇聚到一点时，多值逻辑 VHDL 语言可以构建信号分辨。

已分辨信号具有与之相对应的分辨函数，而未分辨信号没有。我们前面已经使用过未分辨位信号。如果我们在两个并发语句或者两个进程中为一个位信号 *B* 赋两个不同的值，编译时将产生错误，因为编译器无法分辨 *B* 的准确值。

考虑下面的三条并发语句，其中 *R* 为 X01Z 数据类型的已分辨信号：

```
R<= transport '0' after 2 ns, 'Z' after 6 ns;
R<= transport '1' after 4 ns;
R<= transport '1' after 8 ns, '0' after 10 ns;
```

假设 *R* 被初始化为'Z'，如图 8.11 所示，则必须为 *R* 创建三个驱动器。每次未分辨信号 *s*(0), *s*(1) 或 *s*(2)中将有一个发生变化，分辨函数被自动调用来决定已分辨信号 *R* 的值。

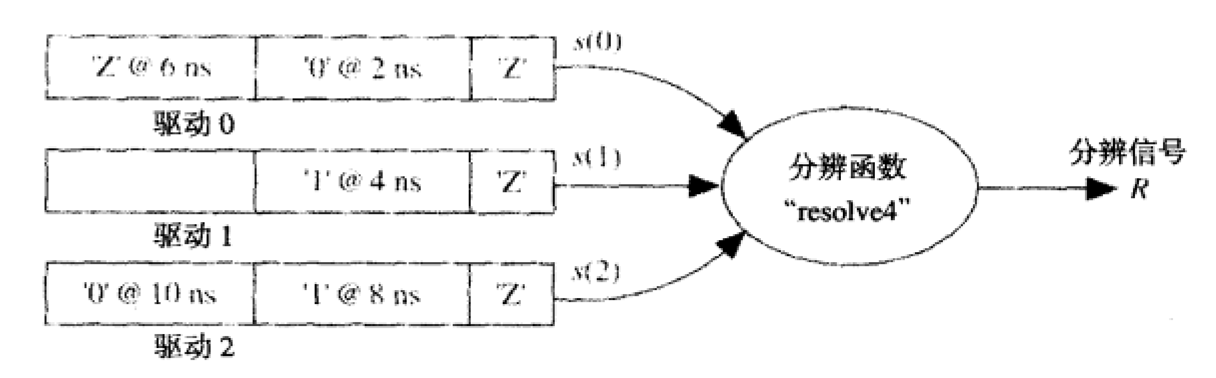


图 8.11 信号驱动器的分辨

由于 X01Z 逻辑中具有表示高阻状态的符号，所以我们可以创建分辨函数对多个连在一起的信号线进行模拟。图 8.12 说明了在一个名为 *fourpack* 的包集合中，X01Z 逻辑的分辨函数是如何被定义的。首先，同时定义了一个未分辨逻辑类型 u_X01Z 和一个与之相对应的不受约束数组类型 u_X01Z_vector。然后，定义说明一个名为 *resolve4* 的分辨函数。已分辨逻辑 X01Z 被定义为 u_X01Z 的子数据类型。子数据类型的定义说明中包含了函数名为 *resolve4* 的函数。这样每当一个数据类型为 X01Z 的信号被计算时，函数 *resolve4* 将被自动调用以决定正确的值。

一个基于三态总线操作的分辨函数的具体设定如下表所示：

	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'X'	'X'
'0'	'X'	'0'	'X'	'0'
'1'	'X'	'X'	'1'	'1'
'Z'	'X'	'0'	'1'	'Z'

对于每对输入值，表中给出了信号的分辨值：如果一个输入值为 Z，则分辨值为另一个输入值；只要有一个输入为 X，则分辨值为 X；如果两个输入值分别为 0 和 1，则分辨值为 X。函数 *resolve4* 中含有一个变量 *s*，它代表一个或多个等待分辨的信号向量。如果向量的长度为 1，那么将返回向量的第一个（也是唯一一个）元素。否则，返回值（已分辨信号）将进行迭代计算，从 *result* = 'Z'开始，然后对表中 *s* 向量的每个元素依次进行查找，并对 *result* 进行重新计算。在图 8.11 的例子中，向量 *s* 有 3 个元素，函数 *resolve4* 在 0, 2, 4, 6, 8 和 10 ns 的时候被调用来计算 *R*。结果如下表所示：

时间	$s(0)$	$s(1)$	$s(2)$	R
0	'Z'	'Z'	'Z'	'Z'
2	'0'	'Z'	'Z'	'0'
4	'0'	'1'	'Z'	'X'
6	'Z'	'1'	'Z'	'1'
8	'Z'	'1'	'1'	'1'
10	'Z'	'1'	'0'	'X'

```
package fourpack is
  type u_x01z is ('X', '0', '1', 'Z');            -- u_x01z is unresolved
  type u_x01z_vector is array (natural range <>) of u_x01z;
  function resolve4 (s: u_x01z_vector) return u_x01z;
  subtype x01z is resolve4 u_x01z;
  -- x01z is a resolved subtype which uses the resolution function resolve4
  type x01z_vector is array (natural range <>) of x01z;
end fourpack;

package body fourpack is
  type x01z_table is array (u_x01z, u_x01z) of u_x01z;
  constant resolution_table: x01z_table := (
    ('X','X','X','X'),
    ('X','0','X','0'),
    ('X','X','1','1'),
    ('X','0','1','Z'));

  function resolve4 (s:u_x01z_vector)
    return u_x01z is

  variable result: u_x01z := 'Z';
  begin
    if (s'length = 1) then
      return s(s'low);
    else
      for i in s'range loop
        result := resolution_table(result, s(i));
      end loop;
    end if;
    return result;
  end resolve4;
end fourpack;
```

图 8.12　X01Z 逻辑的分辨函数

为了使用 X01Z 逻辑来编写 VHDL 代码，我们需要为这种逻辑类型所需的操作进行定义。例如，AND 和 OR 可以用下面的表进行定义：

AND	'X'	'0'	'1'	'Z'
'X'	'X'	'0'	'X'	'X'
'0'	'0'	'0'	'0'	'0'
'1'	'X'	'0'	'1'	'X'
'Z'	'X'	'0'	'X'	'X'

OR	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'1'	'X'
'0'	'X'	'0'	'1'	'X'
'1'	'1'	'1'	'1'	'1'
'Z'	'X'	'X'	'1'	'X'

左边的表对应于 4 值输入 AND 门的工作情况。如果 AND 门的一个输入为'0'，则输出总为 0。如果两个输入都为 1，则输出也为 1。在所有其他的情况下，输出都是未知的('X')，因为高阻门输入既可以为 0 又可以为 1。对于一个 OR 门，如果一个输入是 1，那么输出总是 1。如果两个输入都是 0，则输出也是 0。在所有其他情况下，输出为'X'。基于这两个表的 AND 和 OR 函数可以放入程序包集合 *fourpack* 中用来做 AND 和 OR 的重载操作符。

虽然本节中介绍了如何创建已分辨信号，但幸运地是我们不必创建此类信号。现在已经有提供已分辨信号数据类型的标准库了，例如多值逻辑库 IEEE 1164 和 IEEE_numeric_std。

8.6 IEEE 9 值逻辑系统

IEEE1164 标准定义了一个带有信号分辨的 9 值逻辑系统。标准中定义的 9 个值为

'U'	未初始化
'X'	强未知
'0'	强 0
'1'	强 1
'Z'	高阻
'W'	弱未知
'L'	弱 0
'H'	弱 1
'–'	随意值

其中，未知、0 和 1 这三个值有两种强度：强和弱。强 1 是指信号电压值与提供的电压值相近。弱 1 是指逻辑高电平，但是信号的实际电压有衰减（如具有上拉电阻的输出）。强 0 表示完全接地，弱 0 表示逻辑低电平，但是实际上没有接地（如具有下拉电阻的输出）。9 值系统中用'U'表示未进行初始化的信号，并用'–'表示随意值。

如果一个强信号和一个弱信号被接在一起，则强信号起作用。例如，如果'0'和'H'接在一起，则结果是'0'。这个 9 值逻辑在模拟某些 IC 内部的操作时非常有用。在本节中，我们通常只采用 IEEE 标准值的一个子集——'X', '0', '1', 'Z'。

IEEE-1164 标准为 9 值逻辑定义了 AND, OR, NOT, XOR 以及其他函数。IEEE.双通道 logic_1164 包集合中定义了 std_logic 数据类型，该数据类型使用 9 值逻辑。此标准还定义了很多 9 值逻辑的子数据类型，例如我们前面用过的 X01Z 子数据类型。与位向量相同，当向量用 std_logic 数据类型创建时，它们被称为 std_logic 向量。当使用位向量时，通常把它们的初值赋为 0，但是当使用 std_logic 数据类型时，默认的初始值为'U'。

表 8.5 给出了 IEEE 9 值逻辑的分辨函数表。行索引值被列在表的右边。X01Z 逻辑的分辨函数表是整个表的一部分，在表中用黑框框出。

表 8.5 IEEE 9 值逻辑的分辨函数表

```
CONSTANT resolution_table : stdlogic_table := (
--   ---------------------------------------------------------
--  |  U    X    0    1    Z    W    L    H    -        |   |
--   ---------------------------------------------------------
    ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
    ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
    ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
    ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
    ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
    ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
    ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
   );
```

表 8.6 给出了 IEEE 9 值逻辑的 AND 函数表。行索引值被列在表的右边。X01Z 逻辑的 AND 函数表是整个表的一部分，在表中用黑框框出。IEEE-1164 标准首先定义了 std_ulogic（未分辨标准逻辑）；然后使用相关分辨函数把 std_logic 数据类型定义为 std_ulogic 的子类型。

表 8.6 IEEE 9 值逻辑的 AND 函数表

```
CONSTANT and_table : stdlogic_table := (
--  ---------------------------------------------------------
--  | U    X    0    1    Z    W    L    H    -          |   |
--  ---------------------------------------------------------
    ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
    ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
    ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
    ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
    ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
    ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
    ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
    ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
    ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )  -- | - |
);
```

图 8.13 中的 **and** 函数使用了表 8.6。这些函数提供操作符的重载操作，这意味着如果我们编写一个使用 **and** 函数的语句，则编译器将自动根据操作数的类型来调用适当的 **and** 函数进行求值。如果 **and** 运算的操作数为位变量，则使用普通的 **and** 函数，但是如果操作数的数据类型是 std_logic，则需要调用 std_logic 的 **and** 函数。重载操作符还能够自动给向量提供适当的 **and** 函数。如果操作数是位向量(bit_vector)，则使用通常的方法，按位进行运算，但是如果操作数是 std_logic 向量，那么 std_logic 的 **and** 运算是以按位运算为基础的。图 8.13 中的第一个 **and** 函数通过查表来计算左边(l)和右边(r)操作数 **and** 运算的结果。虽然 **and** 函数首先是针对 **std_ulogic** 定义的，但是由于 std_logic 是 std_ulogic 的子类型，所以 **and** 函数也可以用于 std_logic 数据类型。第二个 **and** 函数对 std_logic 向量进行操作，我们使用了别名以确保两个操作数的索引的取值范围相同。如果向量的长度不同，则由于存在 **assert** 错误，使用显示错误信息，否则，最终结果向量的每一位可以通过查表得到。

```
function "and"  ( l : std_ulogic; r : std_ulogic ) return UX01 is
begin
   return (and_table(l, r));
end "and";

function "and"  ( l,r : std_logic_vector ) return std_logic_vector is
   alias lv : std_logic_vector ( 1 to l'LENGTH ) is l;
   alias rv : std_logic_vector ( 1 to r'LENGTH ) is r;
   variable result : std_logic_vector ( 1 to l'LENGTH );
begin
   if ( l'LENGTH /= r'LENGTH ) then
      assert FALSE
      report "arguments of overloaded 'and' operator are not of the same length"
      severity FAILURE;
   else
      for i in result'RANGE loop
         result(i) := and_table (lv(i), rv(i));
      end loop;
   end if;
   return result;
end "and";
```

图 8.13 std_logic_vector 的 AND 函数

如果需要使用多值逻辑，那么我们需要使用 IEEE. numeric_std 包集合代替之前使用的 numeric_bit 包集合。IEEE. numeric_std 包集合与 numeric_bit 包集合很相似，但是前者把向量的无符号类型和有符号类型作为 std_logic 类型加以定义，而不是定义为位向量；与 numeric_bit 包集合一样，它也为无符号数和有符号数定义了同样的重载操作符和重载函数。

一个 VHDL 程序中要使用无符号类型的向量，如果我们要使用 9 值逻辑下的向量，则只需把

语句

```
use IEEE.numeric_bit.all;
```

替换为语句

```
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
```

IEEE.numeric_std 包集合使用 1164 标准中的 std_logic 类型。因此，这两条语句必须都写。只要写了这两条语句，无符号数据类型就可以使用 9 值逻辑了，程序中的其他部分无需修改。如果原始程序使用位（bit）数据类型，则要把它们转换为 std_logic 类型。

此外，还有一些常用于多值逻辑仿真和综合的 VHDL 包集合，如 std_logic_arith 包集合和 std_logic_unsigned 包集合。这两个包集合都是由 Synopsis 公司开发的。使用这两个包集合时，必须要嵌入以下语句：

```
use IEEE.std_logic_unsigned.all;
use IEEE. std_logic_arith.all;
```

从此例开始，我们就使用 IEEE numeric_std 包集合了，因为它是 IEEE 标准包集合，而且功能与我们之前使用的 numeric_bit 包集合相似。我们不使用 std_logic_arith 和 std_logic_unsigned 包集合，因为它们都不是 IEEE 标准包集合，而且它们的功能没有 IEEE numeric_std 包集合全面。

8.7 基于 IEEE 1164 的 SRAM 模型

本章中，我们介绍如何使用 VHDL 语言对静态 RAM（SRAM）存储器的操作进行描述。RAM 代表随机存储器，意思是存储器中的任何一个字都可以在同一段时间内与其他字一起进行随机存取。严格地说，ROM 存储器也是随机存取的，但是从历史上来看，RAM 这个词通常只用于读-写存储器。此模型还使用了多值逻辑系统，多值逻辑被用来模拟存储数据线的三态条件。

图 8.14 为一个静态 RAM 的框图。此 SRAM 具有 n 条地址线、m 条数据线和 3 条控制线。此存储器可以存储 2^n 个字，且每个字的宽度为 m 位。数据线是双向的，这样可以降低所需的管脚数和存储器晶片的包集合大小。如果从 RAM 中读取数据，则数据线作为输出；如果向 RAM 写入数据，则数据线作为输入。三条控制线的功能如下：

$\overline{CS}$ 如果是低电平，则选中存储器晶片进行读写操作

$\overline{OE}$ 如果是低电平，则选择存储器输出，并输出到外部总线上

$\overline{WE}$ 如果是低电平，则允许数据写入 RAM

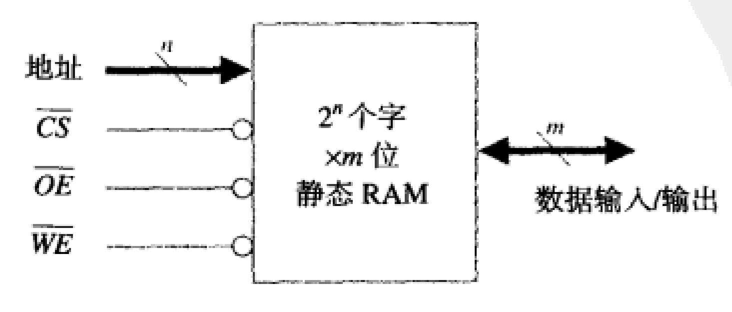


图 8.14 静态 RAM 的框图

当一个信号处于激活状态时，就会对此状态进行说明（assert）。当低电平有效信号处于低电平时，它会被说明；当高电平有效信号处于高电平时，它也会被说明。

RAM 的真值表（表 8.7）描述了它的基本操作。在 I/O 栏中 High-Z 表示输出缓存器的输出为高阻，且没有使用数据输入。在读模式下，地址线进行译码并选择 *m* 个存储单元，在存储器访问时间结束后，数据在 I/O 线上输出。在写模式下，如果 $\overline{WE}$ 是低电平，则数据被路由到选中存储单元的输入锁存器中，但是直到 $\overline{WE}$ 达到高电平或者晶片未被选中时，此写操作才完成。下面的真值表没有考虑存储器的时序。

表 8.7　静态 RAM 的真值表

$\overline{CS}$	$\overline{OE}$	$\overline{WE}$	模式	I/O 管脚
H	X	X	未选中	高-Z
L	H	H	输出不可用	高-Z
L	L	H	读	数据出
L	X	L	写	数据入

现在我们编写一个简单的 VHDL 模块，不考虑时序问题。在图 8.15 中，我们使用由无符号标准逻辑向量(*RAM1*)构成的数组表示 RAM 存储的数组。此存储器可以存储 256 个字，每个字为 8 位。由于 *Address* 的数据类型为无符号位向量，所以为了能够索引存储器数组，它必须转换为整数类型。如果芯片没有被选中，则 RAM 进程就把 I/O 线设置为高 Z。如果 *We_b* = '1'，则 RAM 处于读模式，数据通过 I/O 从存储数组中读出。如果 *We_b* = '0'，则存储器处于写模式，在 *We_b* 上升沿到来时，I/O 线上的数据被写入 *RAM1* 中。如果 *Address* 和 *We_b* 同时改变，则采用 *Address* 的旧地址。*Address'delayed* 为数组的索引，它会延迟 Δ 时间以确保旧地址被使用。在本章前几节中（表 8.3）*Address'delayed* 使用了一个信号属性。此存储器可以进行异步读操作和同步写操作。

```
-- Simple memory model
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity RAM6116 is
  port(Cs_b, We_b, Oe_b: in std_logic;
       Address: in unsigned(7 downto 0);
       IO: inout unsigned(7 downto 0));
end RAM6116;

architecture simple_ram of RAM6116 is
type RAMtype is array(0 to 255) of unsigned(7 downto 0);
signal RAM1: RAMtype := (others => (others =>'0'));
                        -- Initialize all bits to '0'
begin
  IO <= "ZZZZZZZZ" when Cs_b = '1' or We_b = '0' or Oe_b = '1'
    else RAM1(to_integer(Address));    -- read from RAM
  process(We_b, Cs_b)
  begin
    if Cs_b = '0' and rising_edge(We_b) then   -- rising-edge of We_b
      RAM1(to_integer(Address'delayed)) <= IO; -- write
    end if;
  end process;
end simple_ram;
```

图 8.15　简单存储器模块

8.8　SRAM 读/写系统模块

为了进一步说明如何多值逻辑，现在我们讨论一个具有双向三态总线的例子。我们要设计一个存储器读写系统，它可以从 RAM 的 32 个存储单元中读取数据，再对每个数据值加 1，然后把

数据存回 RAM 中。此系统的框图如图 8.16 所示。我们使用一个数据寄存器存储从存储器中读取的字；我们使用一个存储地址寄存器(MAR)存储我们所访问的存储单元的地址。此系统从 RAM 中读取一个字，然后把存储器地址寄存器加 1，并持续此过程直到存储地址等于 32。

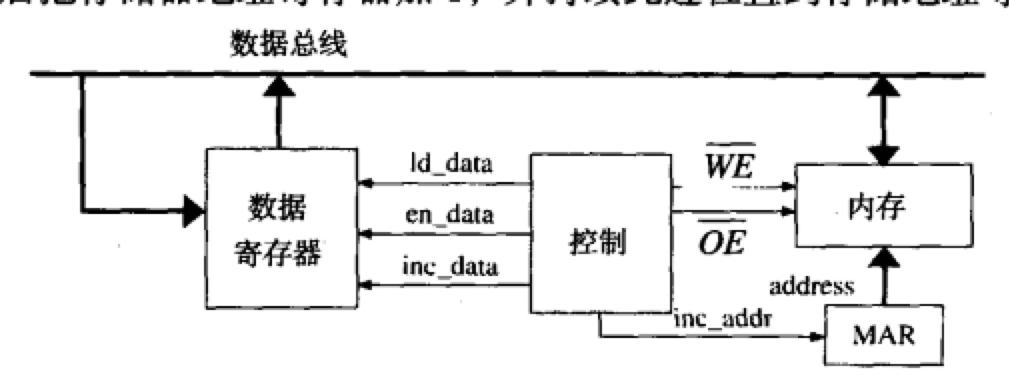


图 8.16　RAM 系统框图

数据总线是双向总线。在进行读操作时，存储器的输出出现在总线上，这时数据寄存器输出到数据总线时，就会处于三态缓冲状态。在进行写操作时，数据寄存器的输出出现在数据总线上，且存储器将其用为输入数据。

操作此系统所需的控制信号：

ld_data	从数据总线载入数据到数据寄存器
en_data	数据寄存器输出到数据总线的使能信号
inc_data	数据寄存器加 1
inc_addr	MAR 加 1
$\overline{WE}$	SRAM 写使能信号
$\overline{OE}$	SRAM 输出使能信号

图 8.17 给出了系统的 SM 图。此 SM 图有 4 个状态。在第 1 个状态，SRAM 驱动存储器中的数据到总线，而且数据被载入到数据寄存器中。此状态中控制信号 $\overline{OE}$ 和 *ld_data* 为真。在状态 S_1，数据寄存器加 1，控制信号 *en_data* 为真，因此数据寄存器驱动总线。写使能信号 $\overline{WE}$ 为低电平有效，并且只有在状态 S_2 时 $\overline{WE}$ 才有效，即 $\overline{WE}$ 在其他状态均为高电平。这样在状态从 S_2 转换为 S_3 时，数据寄存器中的内容写入 RAM 中。存储地址加 1。此过程将一直继续下去直到存储地址等于 32。

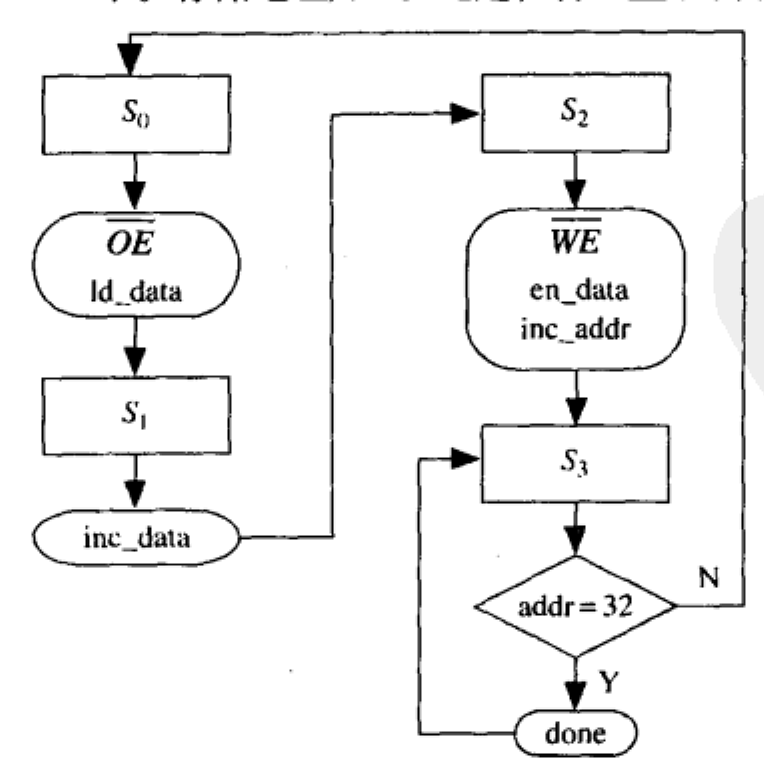


图 8.17　RAM 系统的 SM 图

图 8.18 为 RAM 系统的 VHDL 代码。第一个进程用来表示 SM 图，第二个进程用来在时钟的上升沿更新寄存器。如果地址要增加，则需要另外加入一个短延迟时，以确保在地址变化前可以完成存储器写操作。我们用并发语句对三态缓冲器进行模拟。三态缓冲器可以控制是否把数据寄存器中的数据输出到 I/O 线上。

我们可以对此系统进行修改，使其能够把所有的存储单元都包含在内，用于测试整个 SRAM。存储器系统经常通过在所有的存储单元中写入测试版模式图案（交替的 0 和 1）进行测试。例如，我们可以把 01010101（十六进制表示为 55）写入所有的奇地址中，把 10101010（十六进制表示为 AA）写入所有的偶地址中，然后把奇地址内容和偶地址内容交换。我们把此系统的 VHDL 代码留做习题，请大家在课后完成。

```
-- SRAM Read-Write System model
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity RAM6116_system is
end RAM6116_system;

architecture RAMtest of RAM6116_system is
component RAM6116 is
  port(Cs_b, We_b, Oe_b: in std_logic;
       Address: in unsigned(7 downto 0);
       IO: inout unsigned(7 downto 0));
end component RAM6116;

signal state, next_state: integer range 0 to 3;
signal inc_addr, inc_data, ld_data, en_data, Cs_b, clk, Oe_b, done:
       std_logic := '0';
signal We_b: std_logic := '1';                     -- initialize to read mode
signal Data: unsigned(7 downto 0);                 -- data register
signal Address: unsigned(7 downto 0) := "00000000"; -- address register
signal IO: unsigned(7 downto 0);                -- I/O bus
begin
  RAM1: RAM6116 port map (Cs_b, We_b, Oe_b, Address, IO);
  control: process(state, Address)
  begin
    --initialize all control signals (RAM always selected)
    ld_data <= '0'; inc_data <= '0'; inc_addr <= '0'; en_data <= '0';
    done <= '0'; We_b <= '1'; Cs_b <= '0'; Oe_b <= '1';

    --start SM chart here
    case state is
      when 0 => Oe_b <= '0'; ld_data <= '1'; next_state <= 1;
      when 1 => inc_data <= '1'; next_state <= 2;
      when 2 => We_b <= '0'; en_data <= '1'; inc_addr <= '1'; next_state <= 3;
      when 3 =>
        if (Address = "00100000") then done <= '1'; next_state <= 3;
        else next_state <= 0;
        end if;
    end case;
  end process control;

  --The following process is executed on the rising edge of a clock.
  register_update: process(clk)   -- process to update data register
  begin
    if rising_edge(clk) then
      state <= next_state;
      if (inc_data = '1') then data <= data + 1; end if;
                          -- increment data in data register
      if (ld_data = '1') then data <= IO; end if;
                          -- load data register from bus
      if (inc_addr = '1') then Address <= Address + 1 after 1 ns; end if;
                          -- delay added to allow completion of memory write
    end if;
  end process register_update;

  -- Concurrent statements
  clk <= not clk after 100 ns;
  IO <= data when en_data = '1'
    else "ZZZZZZZZ";
end RAMtest;
```

图 8.18　RAM 系统的 VHDL 代码

8.9 类属语句

类属通常用于设定一个元件的参数，这样在进行元件例化时就可以对参数的值进行设定了。例如，一个门的上升沿时刻和下降沿时刻可以用 generic 进行指定，并且 generic 语句可以把门的每个例化都赋为不同的值。图 8.19 为一个两输入 NAND 门的 VHDL 程序。此 NAND 门的上升沿和下降沿的延迟时间均取决于门上负载的数量。在实体说明中，*Trise*, *Tfall* 和 *load* 均为类属，且分别代表无负载上升时刻、无负载下降时刻和负载数量。在构造体中，每当 *a* 或 *b* 变化时，内部的 *nand_value* 都会被计算。如果刚好在 *nand_value* 变化为'1'，输出出现一个上升沿，则门延时为

$$Trise + 3\ \text{ns} \times load$$

其中，3 ns 为每个负载的附加延时。否则，如果刚好在 *nand_value* 变化为'1'时，输出出现一个下降沿，则门延时为

$$Tfall + 2\ \text{ns} \times load$$

其中，2 ns为每个负载的附加延时。

```
entity NAND2 is
  generic(Trise, Tfall: time; load: natural);
  port(a, b: in bit;
       c: out bit);
end NAND2;

architecture behavior of NAND2 is
signal nand_value: bit;
begin
  nand_value <= a nand b;
  c <= nand_value after (Trise + 3 ns * load) when nand_value = '1'
    else nand_value after (Tfall + 2 ns * load);
end behavior;

entity NAND2_test is
  port(in1, in2, in3, in4: in bit;
       out1, out2: out bit);
end NAND2_test;

architecture behavior of NAND2_test is
component NAND2 is
  generic(Trise: time := 3 ns; Tfall: time := 2 ns; load: natural := 1);
  port(a, b: in bit; c: out bit);
end component;
begin
  U1: NAND2 generic map (2 ns, 1 ns, 2) port map (in1, in2, out1);
  U2: NAND2 port map (in3, in4, out2);
end behavior;
```

图 8.19　用类属语句模拟上升/下降时刻

实体 *NAND2_test* 用来对元件 NAND2 进行测试。在构造体的元件说明中，*Trise*, *Tfall* 和 *load* 的初值为默认值。当 *U1* 被例化时，类属映射为 *Trise*, *Tfall* 和 *load* 指定了不同的值，当 *U2* 被例化时没有类属映射，所以使用默认值。

8.10　命名关联

在例化语句的端口映射和类属映射中，我们使用了**位置关联**(positional association)这一概念。例如，假设一个全加器的实体说明为

```
entity FullAdder is
  port(X, Y, Cin: in bit; Cout, Sum: out bit);
end FullAdder
```

语句

```
FA0:  FullAdder port map (A(0), B(0), '0', open, S(0));
```

生成了一个全加器，并且把 *A*(0)同加法器的输入 *X* 关联，*B*(0)和输入 *Y* 关联，'0'与输入 *Cin* 关联，输出 *Cout* 没有关联，*S*(0)与加法器输出 *Sum* 关联。端口映射中的第一个信号与实体说明中第一个信号相关，相对应的第二个信号也相关，依此类推。为了指出没有关联的地方，我们使用了关键词 **open**。

同样，我们也可以使用命名关联，就是把端口映射中的每个信号与元件实体说明中的每个信号关联。例如，语句

```
FA0:  FullAdder  port map (Sum =>S(0), X => A(0), Y =>B(0), Cin => '0');
```

与前面的例化语句具有相同的关联（如 *Sum* 与 *S*(0)关联，*X* 与 *A*(0)关联，等等）。当使用命名关联时，端口映射中各个关联的排列顺序不重要，而且没有列出的端口信号都没有关联。命名关联的使用提高了代码的易读性，并且为信号的排列提供了灵活性。

当命名关联在一个类属映射中使用时，任何没有被关联的类属变量都赋予默认值。例如，如果我们把图 8.19 中的标号 *U*1 用下面的语句取代，则

```
U1: NAND2 generic map( load =>3, Trise => 4 ns) port map(in1, in2, out1);
```

则 *Tfall* 会被赋予默认值 2 ns。

8.11　生成语句

在第 2 章中，我们介绍了 4 个全加器元件的例化语句，并且把它们连接起来构成了 1 个 4 位加法器。如果加法器为 8 位或者更多位，那么要是为每种情况都指定端口映射，就会很烦琐。如果需要重复使用一个列相同的元件，那么生成语句为这些元件的例化提供了一个简单的方法。图 8.20 说明了如何使用生成语句对四个 1 位加法器进行元件例化，使它们构成一个 4 位加法器。我们用一个 5 位向量代表进位，*Cin* 与 *C*(0)相同，*Cout* 与 *C*(4)相同。**for** 循环生成了全加器的四个拷贝，每个拷贝中都有恰当的 port map，并且它们还设定了加法器之间的连接。

另外，在设计阵列乘法器时，生成语句也非常有用。阵列乘法器（第 4 章）的 VHDL 代码中重复使用了 port map 语句对每个元件进行例化。我们可以使用生成语句代替这些语句。

在前面的例子中，我们使用的生成语句的格式为

```
标识名:   for 标识符 in 取值范围 generate
```

```
[begin]
   并发语句
end generate [标识名];
```

在编译时，为了在给定范围内生成每个标识符的值，所以生成一列并发语句。在图 8.20 中使用了一条并发语句——元件例化语句。生成语句本身就是一条并发语句，因此生成语句允许嵌套使用。

```
entity Adder4 is
   port (A, B: in bit_vector(3 downto 0); Ci: in bit;      -- Inputs
      S: out bit_vector(3 downto 0); Co: out bit);         -- Outputs
end Adder4;

architecture Structure of Adder4 is
component FullAdder
   port (X, Y, Cin: in bit;        -- Inputs
      Cout, Sum: out bit);         -- Outputs
end component;

signal C: bit_vector(4 downto 0);

begin
   C(0) <= Ci;
      -- generate four copies of the FullAdder
   FullAdd4: for i in 0 to 3 generate
   begin
      FAx: FullAdder port map (A(i), B(i), C(i), C(i+1), S(i));
   end generate FullAdd4;
   Co <= C(4);
end Structure;
```

图 8.20　使用生成语句的 Adder4

8.11.1　条件生成语句

带有 if 子句的生成语句可以有条件地生成一系列的并发语句。此类并发语句的格式如下所示：

```
标识名:  if 条件 generate
[begin]
  并发语句
end generate [标识名];
```

在这种情况下，只有当条件为真时，这些并发语句才能在编译时被生成。

图 8.21 给出了如何使用带有 if 子句的生成语句进行条件编译。如果 *Lshift* 为真，则下列语句可以创建一个 *N* 位左移寄存器：

```
genLS: if Lshift generate
   shifter <= Q(N-1 downto 1) & Shiftin;
end generate;
```

如果 *Lshift* 为假，则另一个条件生成语句将创建一个右移寄存器。此例还展示了如何联合使用类属语句和生成语句。从此例中，我们使用类属参数编写 VHDL 模块中的参数，这样模块的大小和功能可以在例化时进行更改。

```
entity shift_reg is
  generic(N: positive := 4; Lshift: Boolean := true);-- generic parameters used
  port(D: in bit_vector(N downto 1);
       Qout: out bit_vector(N downto 1);
       CLK, Ld, Sh, Shiftin: in bit);
end shift_reg;

architecture SRN of shift_reg is
signal Q, shifter: bit_vector(N downto 1);
begin
  Qout <= Q;
  genLS: if Lshift generate      -- conditional generate of left shift register
```

图 8.21　使用条件编译实现的移位寄存器

```
    shifter <= Q(N-1 downto 1) & Shiftin;
  end generate;
  genRS: if not Lshift generate -- conditional generate of right shift register
    shifter <= Shiftin & Q(N downto 2);
  end generate;
  process(CLK)
  begin
    if CLK'event and CLK = '1' then
      if LD = '1' then Q <= D;
      elsif Sh = '1' then Q <= shifter;
      end if;
    end if;
  end process;
end SRN;
```

图 8.21（续）　使用条件编译实现的移位寄存器

8.12　文件和文本输入输出

测试大型 VHDL 设计时，输入文件和文本的能力是很重要的。这节介绍了 VHDL 的文件输入和输出。文件通常和测试平台一起使用，提供测试数据的来源和测试结果的保存。VHDL 提供了一个标准的文件输入输出（TEXTIO）包集合，可以从文件中按行读写文本。

使用文件前，必须进行说明，说明语句格式为

```
file 文件名: 文件类型 [open mode] is "文件路径";
```

例如，语句

```
file test_data: text  open read_mode  is "C:\test1\test.dat"
```

说明了一个名为 *test_data* 的文件，它在读模式下打开。文件的物理存储单元在 C 盘的 *test1* 目录中。

文件可以以 *read_mode*, *write_mode* 或者 *append_mode* 模式打开。在 *read_mode* 下，文件中的字符序列可以用读过程进行读取。当文件在 *write_mode* 下被打开时，计算机的文件系统就会创建一个新的空文件，并且可以用写过程把数据序列写入到该文件中。如果要向一个已有的文件中写入数据，则文件必须在 *append_mode* 下打开。

一个文件只能包含一种类型的对象，例如整数、位向量或文本字符串，这是由文件的类型设定的。例如，说明语句

```
type bv_file is file of bit_vector;
```

定义了 *bv_file* 为一个只包含位矢量的文件类型。每个文件类型都有一个与其相关的隐式文件结束函数，其调用格式为

```
endfile(文件名)
```

如果文件指针在文件的末尾，则此语句返回“真”。

VHDL 的标准 TEXTIO 包集合中包含对文件进行操作的定义说明和过程，这些文件都是由文本行构成的。TEXTIO 包集合（参见附录 C）定义一个名为 text 的文件类型：

```
type text is file of string;
```

TEXTIO 包集合中既包含从 text 类型的文件中按行进行文本读取的过程，也包含按行在文件中写入文本的过程。

过程 readline 读取了一行文本，并将其放置在一个带指针的缓存器中。缓存器的指针必须为

line 类型，其在 TEXTIO 包集合中的说明为

```
type line is access string;
```

只要说明了一个 line 类型的变量，就会生成一个指向字符串的指针。代码

```
variable buff: line;
...
readline(test_data ,)buff;
```

从 *test_data* 中读取一行文本，并将其置于被指针 *buff* 指向的缓存器中。在缓存器中读入一行文本后，我们必须一次或多次调用 read 过程，以便从行缓存器中把数据释放出来。TEXTIO 包集合提供的重载 read 过程可以从缓存器中读取数据类型为位、位向量、布尔量、字符、整数、实数、字符串和时间的数据。例如，如果 *bv4* 是一个长度为 4 的位矢量，则调用

```
read (buff, bv4);
```

可以从缓存器中释放一个 4 位的矢量，把 *bv4* 设置为此向量，并且调整指针 *buff*，使其指向缓存器的下一个字符。再次调用 read 过程可以从行缓存器中释放下一个数据对象。

read 的调用格式可以是以下两种格式中的一种：

```
read (pointer, value);
read (pointer, value, good);
```

其中 *pointer* 为 line 类型，*value* 是我们想读取的数据变量。在第二种格式中，*good* 为布尔量，它在读取成功时返回 TRUE，失败时返回 FALSE。*value* 的大小和类型决定调用 TEXTIO 包集合中的哪一个 read 过程。例如，如果 *value* 为一个长度为 5 的字符串，那么调用 read 后可以从行缓存器中读取下 5 个字符。如果 *value* 的类型是整数，那么调用 read 后可以越过所有的空格对十进制数字进行读取，直到再遇到空格或其他的非数字的字符为止，并且把读取到的字符串转换为整数。在 text 类型的文件中，字符、字符串和位矢量不用符号分隔。

为了向文件中写入文本行，我们必须一次或者多次调用一个 write 过程向行缓存器中写入数据，然后调用 writeline 向文件中写入数据。TEXTIO 包集合提供的重载 write 过程可以把类型为位、位向量、布尔量、字符、整数、实数、字符串和时间的数据写入缓存器。例如，代码

```
variable buffw: line;
variable int1: integer;
variable bv8: bit_vector(7 downto 0);
...
write (buffw, int1, tight, 6);
write (buffw, bv8, right, 10);
writeline (buffw, output_file);
```

可以将 *int1* 转化为文本串，向 *buffw* 指向的行缓存器中写入此字符串，并且调整指针，此文本在一个宽度为 6 个字符的区域内向右对齐。第二个写调用将位向量 *bv8* 置于一个行缓存器中，然后调整指针。这个 8 位矢量在宽度为 10 个字符的区域内向右对齐，然后 writeline 把缓存器的内容写入 *out put_file* 中。每个写调用均有 4 个参数：(1)line 类型的缓存器指针；(2)一个可接收的数据类型的值；(3)在输出区域内指定文本位置的对齐方式（左、右）；(4)区域宽度，一个整数，它设定了区域中字符的个数。

作为一个例子，我们编写了一个从文件中读取数据的过程，并且将数据存储在一个存储器数组中。此过程过后将用于向计算机系统的存储器模块中加载指令代码，然后让此计算机系统执行所存储的指令，并以此对其进行测试。文件中数据的格式如下所示：

```
address N comments
byte1 byte2 byte3...byteN comments
```

地址由 4 个 16 进制数组成，*N* 为整数，它代表下一行代码的字节数。每个字节的代码由 2 个 16 进制数组成。每个字节由一个空格隔开，并且最后一个字节后面必须有一个空格。此空格后面的任何数据都不会被读取，只是被看做为注释。第一个字节应该被存储在给定地址的存储器数组中，第二个字节放在下一个地址中，依此类推。例如，考虑下面的文件：

```
12AC 7 (7 hex bytes follow)
AE 03 B6 91 C7 00 0C
005B 2  (2 hex bytes follow)
01 FC<space>
```

当使用此文件作为输入的 *fill_memory* 过程被调用时，AE 被存储在 12AC，03 存储在 12AD，B6 存储在 12AE，91 存储在 12AF，依此类推。

图 8.22 给出了一个 VHDL 代码，它调用了 *fill_memory* 过程，使其读取文件中的数据，并且将数据存储在一个名为 *mem* 的数组中。因为 TEXTIO 包集合中不包含针对 16 进制数的读取过程，所以 *fill_memory* 过程将每个 16 进制数都看做是字符串，然后将字符串转换为整型。单个的 16 进制数到整型的转换通过查表完成。常数 *lookup* 是一个整型数组，其索引的取值范围为‘0’到‘F’。此取值范围中包含 23 个 ASCII 字符：‘0’, ‘1’, ‘2’, …, ‘9’, ‘:’, ‘;’, ‘<’, ‘ = ’, ‘>’, ‘?’, ‘@’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’。相应的值为 0, 1, 2, …, 9, –1, –1, –1, –1, –1, –1, –1, 10, 11, 12, 13, 14, 15。其中，由于索引中的 7 个特殊字符在实际中根本不会出现，所以–1 可以用任何整数值代替。这样，*lookup*(‘2’)代表整数值 2，*lookup*(‘C’)代表 12，依此类推。

```
library IEEE;
use IEEE.numeric_bit.all;                  -- to use TO_UNSIGNED(int, size)
use std.textio.all;

entity testfill is
end testfill;

architecture fillmem of testfill is
type RAMtype is array (0 to 8191) of unsigned(7 downto 0);
signal mem: RAMtype := (others => (others => '0'));

procedure fill_memory(signal mem: inout RAMType) is
type HexTable is array (character range <>) of integer;
-- valid hex chars: 0, 1, . . . A, B, C, D, E, F (upper-case only)
constant lookup: HexTable('0' to 'F'): =
  (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -1, -1, -1,
   -1, -1, -1, -1, 10, 11, 12, 13, 14, 15);
file infile: text open read_mode is "mem1.txt"; -- open file for reading
-- file infile: text is in "mem1.txt";          -- VHDL '87 version
variable buff: line;
variable addr_s: string(4 downto 1);
variable data_s: string(3 downto 1);            -- data_s(1) has a space
variable addr1, byte_cnt: integer;
variable data: integer range 255 downto 0;
begin
  while (not endfile(infile)) loop
    readline(infile, buff);
    read(buff, addr_s);                         -- read addr hexnum
    read(buff, byte_cnt);                       -- read number of bytes to read
```

图 8.22　从文件读取数据并存储在存储数组中的 VHDL 代码

```
      addr1 := lookup(addr_s(4)) * 4096 + lookup(addr_s(3)) * 256
               + lookup(addr_s(2)) * 16 + lookup(addr_s(1));
      readline(infile, buff);
      for i in 1 to byte_cnt loop
        read(buff, data_s);                    -- read 2 digit hex data and a space
        data := lookup(data_s(3)) * 16 + lookup(data_s(2));
        mem(addr1) <= TO_UNSIGNED(data, 8);
        addr1:= addr1 + 1;
      end loop;
    end loop;
  end fill_memory;
begin
  testbench: process
  begin
    fill_memory(mem);
    -- insert code which uses memory data
  end process;
end fillmem;
```

图 8.22（续） 从文件读取数据并存储在存储数组中的 VHDL 代码

在过程 *fill_memory* 中，调用 *readline* 从文本中读取一行数据，此行文本为一个 16 进制地址和一个整数。第一个读调用从行缓存器中读取地址字符串，第二个读调用读取表示下一行字节数的整数。通过对地址字符串中每个字符进行查表，我们可以计算得到整数 *addr1*。文本的下一行被读入到缓存器中，我们使用一个循环语句对地址串中的每个字节进行读取。因为 *data_s* 的长度是三个字符，所以每个读调用将读取两个 16 进制数和一个空格。存储在存储器数组中的 16 进制字符先被转换为整数，然后被转换为 std_logic_vector。在读取和存储下一个字节前，要对地址加 1。到达文件的末尾时，退出过程。

本章介绍了 VHDL 的一些重要特性。首先介绍了函数和过程语句，接着讨论了属性语句。使用信号属性语句，我们可以检测建立时间和保持时间，以及其他的时序特征。使用数组属性语句，我们可以写出不依赖于数组索引的过程。重载操作符可以扩展 VHDL 操作符的定义，从而我们可以把这些操作符应用于不同类型的操作数。IEEE 标准 1164 定义了一个广泛使用于 VHDL 的 9 值逻辑系统。多值逻辑和与之相关的分辨函数使我们可以对三态总线和其他的多源信号驱动系统进行模拟。类属语句使我们可以在一个元件被例化时，为它指定一个参数值。生成语句为我们提供了一个描述具有迭代结构系统的高效方法。TEXTIO 包集合为文件的输入和输出提供了一个方便的方法。

习题

8.1 编写一个 VHDL 函数，它可以把一个 5 比特位向量转换为一个整数。注意二进制数 $a_4a_3a_2a_1a_0$ 的整数值按下面的方法计算：$((((0+a_4)\times2+a_3)\times2+a_2)\times2+a_1)\times2+a_0$。你所编写的函数需要的仿真时间为多少？

8.2 写出计算 *n* 位向量补码的 VHDL 函数。函数调用的格式为 `comp2(bit_vec, N)`，其中 *N* 表示向量的长度，并要求写出你对 bit_vec 的取值范围所作的所有假设。要求使用一个循环按位进行取补。

8.3 编写一个 VHDL 函数，它可以返回 *N* 个整数中的最大值。函数调用的格式为 `LARGEST(ARR, N)`。

8.4 *A* 和 *B* 是表示无符号二进制数的位向量。编写一个 VHDL 函数，当 $A>B$ 时，返回 TRUE。函数调用格式为 GT(A, B, N)，其中 *N* 为向量的长度。在你所编写的函数中不可以调用任何函数或过程。提示：首先比较 *A* 和 *B* 的最高有效位，再从左到右一位一位地进行比较。例如，

如果 $A = 1011010, B = 1010110$，当比较到第四位时，就可以确定 $A > B$。

8.5 VHDL 语言中函数与过程的主要区别是什么？

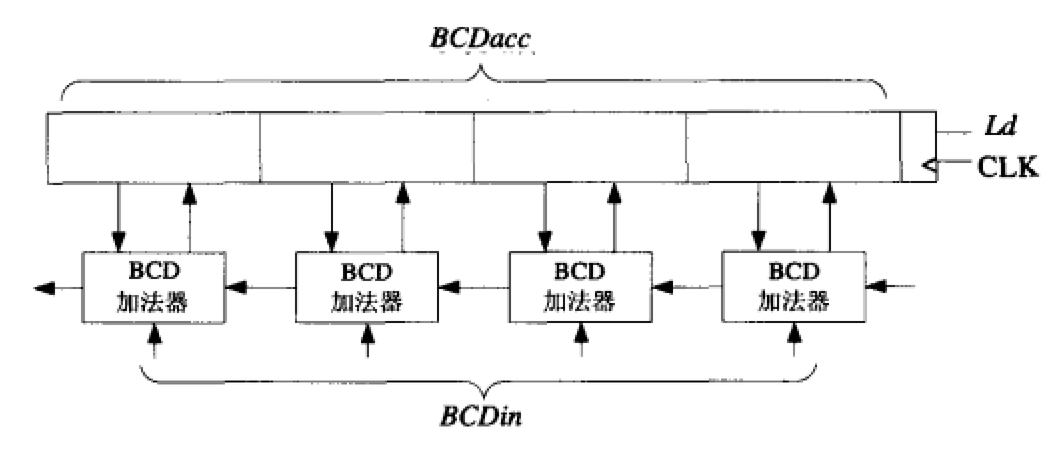
8.6 编写一个 VHDL 过程，它可以计算一个 N 位输入向量中 1 的个数($N \leqslant 31$)。输出为 5 位无符号向量。

8.7 X 和 Y 是长度为 N 的位向量，用以表示有符号二进制数（用补码表示负数）。写出 $D = X-Y$ 的 VHDL 过程。要求此过程能返回最后一位(B)的借位及溢出标志(V)。此过程中不允许调用任何其他函数，过程调用的格式为 SUBVEC(X, Y, D, B, V, N);。

8.8 写出一个带累加器的 4 数字 *BCD* 加法器（框图如下）的 VHDL 程序。如果 $LD = 1$，则 *BCDacc* 中的内容被 *BCDacc* + *BCDin* 所代替。每个 4 数字 BCD 信号均要求使用一个数组表示，其格式为

```
type BCD4 is array (3 downto 0) of unsigned (3 downto 0);
```

写出一个过程语句，它可以计算两个 BCD 数字与一个进位的和，并返回一个 BCD 数字和一个进位。在你所编写的代码中要求调用此过程语句 4 次。



8.9 观察下面的 VHDL 代码，列出每次发生变化产生时，B 和 C 的值，直到时间大于 8 ns 时（Δ 时间考虑在内）结束。假设 B 在 5 ns 时变为"0110"，请指出在何时刻过程 $P1$ 被调用。

```
entity Q1 is
  port(B, C: inout bit_vector(3 downto 0));
end Q1;

architecture Q1 of Q1 is
  procedure P1(signal A: inout bit_vector) is
  begin
    for i in 1 to 3 loop
      A(i) <= A(i-1);
    end loop;
    A(0) <= A(3);
  end P1;
 begin
   process
   begin
     wait until B'event;
     P1(B);
     wait for 1 ns;
     P1(B);
   end process;
   C <= B;
 end Q1;
```

8.10 下面的 VHDL 代码是一个进程的一部分。假设在代码执行前，$A = B =$ '0'。请给出在代码刚刚执行时，变量 X_1, X_2, X_3, X_4 的值。

```
wait until clock'event and clock = '1';
A <= not B;
A <= transport B after 5 ns;
wait for 5 ns;
X1 := A'event;
X2 := A'delayed'event;
X3 := A'last_event;
X4 := A'delayed'last_event;
```

8.11 写出计算两个整数向量 *A*, *B* 点乘($C=\sum a_i \times b_i$)的 VHDL 函数程序。函数调用语句的格式为 DOT(A, B)，其中 *A* 和 *B* 为整数向量信号。使用函数内部的属性语句设定矢量的长度和取值范围，不对范围的最大值和最小值作任何假设。例如，

$A(3 \text{ downto } 1) = (1, 2, 3), B(3 \text{ downto } 1) = (4, 5, 6), C = 3\times6+2\times5+1\times4=32$

如果两个数的取值范围不同，则输出一个警告。

8.12 编写一个 VHDL 过程，它可以计算两个 $n\times m$ 维整数矩阵的和，$C <= A+B$。过程调用格式为 addM(A, B, C)。当 *A* 和 *B* 的行数或列数不同时，返回一个错误信号。不要对取值范围的最大最小值和取值方向作任何假设。

8.13 写出一个 VHDL 过程，它可以计算两个表示有符号二进制数的位向量的和。负数用二进制补码表示。如果向量的长度不同，则在求和过程中对较短的向量的符号位进行扩展。不对向量的取值范围作任何假设。过程调用的格式为 Add2(A, B, Sum, V)，如果生成二进制补码溢出，则 $V=1$。

8.14 一个 VHDL 实体中定义了输入 *A*,*B* 和输出 *C*, *D*。*A*, *B* 初始为高电平。当 *A* 变为低电平时，*C* 在 5 ns 后变为高电平；如果 *A* 再次发生变化，则 *C* 在 5 ns 后再次发生变化。若 *B* 在 *A* 改变后 3 ns 内没有发生变化，则 *D* 发生变化。

(a) 写出 VHDL 程序的构造体部分，要求包含一个定义 *C*, *D* 的进程。

(b) 写出另一个进程，检测 *B* 是否在 *A* 变为高电平的前 2 ns 和后 1 ns 内稳定，如果 B 为低电平的时间不足 10 ns，则此进程还要返回一个错误信号。

8.15 写出用于位向量操作符“<”的重载函数语句。当 *A* 小于 *B* 时，返回布尔量 TRUE；否则返回 FALSE。当两位向量的位数不同时，返回错误信号。

8.16 写出用于位向量操作符“-”的重载函数语句。若 *A* 为一个位向量，则“–A”应返回 *A* 的二进制补码。

8.17 考虑下面的三条并行语句，其中 *R* 是 X01Z 型已分辨信号。

```
R <= transport '0' after 2 ns, 'Z' after 8 ns;
R <= transport '1' after 10 ns;
R <= transport '1' after 4 ns, '0' after 6 ns;
```

画出上面语句生成的各个驱动，并给出从 0 ~ 12 ns 内的已分辨输出信号 *R*。

8.18 写出一个地址译码器的 VHDL 程序。此地址译码器的一个输入为 8 位的地址，它的取值范围可以是长度为 8 的任意范围，如 bit_vector addr(8 **to** 15)。第二个输入为 check: x01z_vector(5 **downto** 0)。当 8 位地址向量的前 6 位与 6 位 check 向量相对应时，此地址译码器输出 Sel = ‘1’。例如，如果 *addr* = “10001010”, *check* = “1000XX”, 则 *Sel* = ‘1’。只需比较 *addr* 的最左边 6 位，其余位可忽略。*Check* 向量中的 *X* 表示随意值。

8.19 写出 74HC374（具有 3 状态输出的 8 进制 D 型触发器）中一个触发器的 VHDL 程序。使用 IEEE 标准 9 值逻辑包集合。假设所有的逻辑值为‘x’, ‘0’, ‘1’或‘z’。用 assert 语句检测启动时间，保持时间和脉冲宽度。当 *CLK* 或 *OC* 是‘x’，或触发器中已经存储了一个‘x’ 时，如果输

出不是'z'，则输出一定是'x'。

8.20 编写比较两个 IEEE std_logic 矢量是否相等的 VHDL 函数。如果两向量中出现任何不是'0', '1'或'–'（随意值）的位时，或是出现两个向量的长度不同时，则报错。函数调用时只可以传递向量。当两向量相同时返回 TRUE，不同时返回 FALSE。比较时假设'0' = '–', '1' = '–'。不对两向量的取值范围作任何假设（例如，一个向量可以是 1 to 7，另一个向量可以是 8 downto 0）。

8.21 考虑下面的 VHDL 并行语句，其中 A, B, C 都是 std_logic 类型。

```
A <= transport '1' after 5 ns, '0' after 10 ns, 'Z' after 15 ns;
B <= transport '0' after 4 ns, 'Z' after 10 ns;
C <= A after 6 ns;
C <= transport A after 5 ns;
C <= reject 3 ns B after 4 ns;
```

(a) 画出信号 A, B 的驱动器（见图 2.27）。

(b) 画出 C 的三个驱动 $s(0)$, s(1)和 $s(2)$（类似于图 8.11）。

(c) 写出驱动器分辨后每个时刻的 C 的值，并画出 C 的时序图。

8.22 std_logic 的子类型 X01LH 的值为'X', '0', '1', 'L'和'H'。根据此子类型中的一个分辨函数完成下表。

	'X'	'0'	'1'	'L'	'H'
'X'					
'0'					
'1'					
'L'					
'H'					

8.23 写出"not"函数的重载函数，其中输入和返回值均为 standard logic vector。"not"函数基本上用来模拟一组反相器。输出位为'U', '0', '1'或'X'中的一个。未初始化的输入将给出未初始化的输出。

8.24 在下面代码中，所有信号均为 1 位 std_logic。请画出与此程序相对应的逻辑框图。假设 D 触发器具有 CE 端。

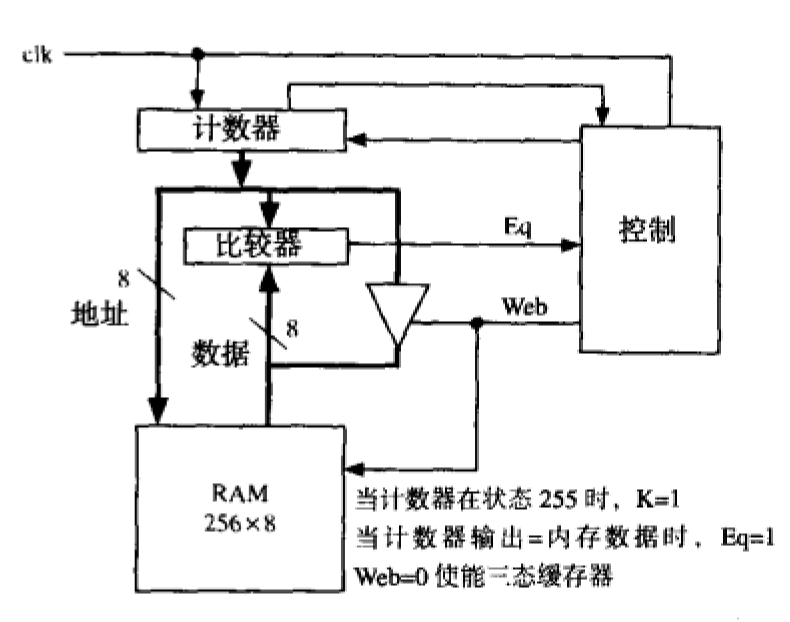
```
F <= A when EA = '1' else B when EB = '1' else 'Z';
process(CLK)
begin
  if CLK'event and CLK = '1' then
    if Ld = '1' then A <= B; end if;
    if Cm = '1' then A <= not A; end if;
  end if;
end process;
```

8.25 设计一个存储器测试系统用以测试一个静态 RAM 存储器的前 256 字节。系统由一个简单的控制器、一个 8 位计数器、一个比较器和一个存储器组成（参见下图）。计数器既与地址相连，也与数据（IO）总线相连，所以 0 被写入 0 号地址线，1 被写入 1 号地址线，2 被写入 2 号地址线……255 被写入 255 号地址线。然后数据将从 0 号地址线、1 号地址线、……255 号地址线中被读出，并与地址线号码做比较，一旦被检测出数据不匹配，则控制器马上变为失败状态。当所有 256 个数据均匹配时，控制器进入通过状态。假设 $OE_b = 0$，$CS_b = 0$。

(a) 画出控制器的 SM 图或状态图（5 个状态）。假设时钟周期足够长，且每个时钟周期内可以读取一个字。

(b) 写出此存储器测试系统的 VHDL 代码。

8.26 设计一个与习题 8.25 相似的存储器测试系统，但是要求在存储器中写入一组测试模板图案（01010101 写入 0 地址，10101010 写入 1 地址等等）。画出框图和 SM 图。



8.27 设计一个存储器检测器以验证 6116 静态 RAM（图 8.15）是否能正常工作。此测试器应能在所有的存储单元中存储测试模板图案（使偶数号地址线中 0, 1 交替出现，使奇数地址线中 1, 0 交替出现），然后将其再读回，接着测试器应该能够使用反向的测试模板图案重新进行一次测试。

(a) 画出存储器测试器的框图。写出并解释所有的控制信号。

(b) 画出控制单元的 SM 图或状态图。假设使用简易 RAM 模块并忽略时序。

(c) 写出此测试器的 VHDL 程序，并编写一个测试程序验证其操作。

8.28 一个时钟控制的触发器从时钟 *CLK* 上升沿到 *Q* 和 *Q'* 发生改变的传输延迟为：如果 $Q(Q')$ 变为 1，则 $t_{plh} = 8$ ns；如果 $Q(Q')$ 变为 0，则 $t_{phl} = 10$ ns。时钟脉冲的最小宽度 $t_{ck} = 15$ ns，输入 *T* 的启动时间 $t_{su} = 4$ ns，持续时间 $t_h = 2$ ns。写出此触发器的 VHDL 程序，且当有任何时序特性没有得到满足时，程序报错。在此程序中使用 generic 参数（使用默认值）。

8.29 (a) 为一个编写具有直接清零输入的 D 触发器编写程序模块。时间参数如下：t_{plh}, t_{phl}, t_{su}, t_h, t_{cmin}。允许的最小时钟周期为 t_{cmin}。违反时序时要求报错。

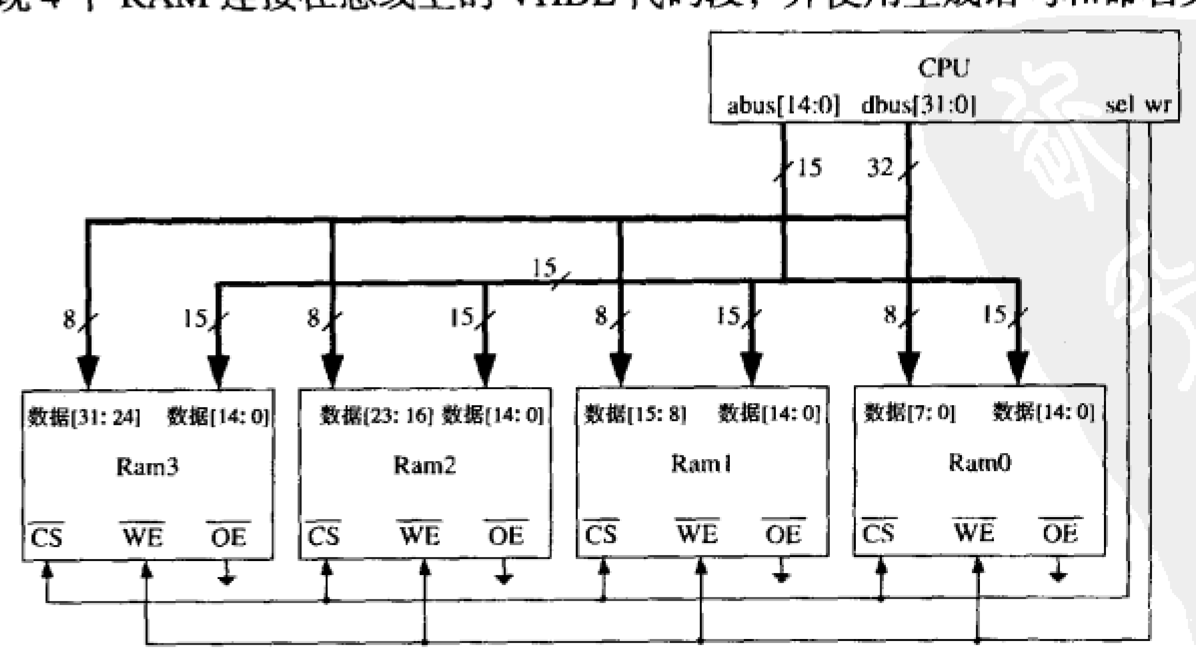
(b) 为此模块编写一个测试程序，要求包括对所有错误情况的测试。

8.30 使用迭代电路编写一个 *N* 位比较器的 VHDL 程序。在实体中，用类属参数 *N* 定义输入位向量 *A* 和 *B* 的长度。当 $A = B$ 时，比较器输出 *EQ* = '1'；当 $A > B$ 时，比较器输出 *GT* = '1'。从最高位开始进行比较，用循环语句进行逐位比较。逐位比较完毕后，*EQ* 和 *GT* 的最后值返回给 *A* 和 *B*。

8.31 4 个 RAM 存储器与 CPU 总线相连，如下所示。假设可以使用下面的 RAM 元件：

```
component SRAM
  port(cs_b, we_b, oe_b: in bit;
       address: in bit_vector(14 downto 0);
       data: inout std_logic_vector(7 downto 0));
end component;
```

写出实现 4 个 RAM 连接在总线上的 VHDL 代码段，并使用生成语句和命名关联。



8.32 写出 N 位串入/串出右移寄存器的结构描述方式 VHDL 代码。此移位寄存器输入为位信号：*SI*（串行输入）、*Sh*（移位使能）和 *CLK*。程序实体说明中应包含一个类属语句，在构造体中也应有一个类属语句。程序中可以把带时钟使能(*CE*)的 D 触发器作为元件。

8.33 写出一个模块的结构描述方式 VHDL 程序，此模块有两个输入：N 位向量 A 和控制信号 B（1 位）。当 $B = 1, C <= A$ 时，模块输出一个 N 位向量 C；当 $B = 0$ 时，输出 C 全为 0。使用 generic 语句设定 N 的值（默认值为 4）。实现此逻辑模块，并使用例化语句设定 N 个 2 输入 AND 门。

8.34 用生成语句实现一个 4×4 阵列乘法器。程序中可以使用第 4 章中介绍过的全加器、半加器和 AND 门作为元件。

8.35 B 为一个整数数组，其取值范围为 0 to 4。编写一个 VHDL 程序段，它可以从一个名为"FILE2"的文件中读取一行文本，然后把读取的 5 个整数写入数组 B 中。假设 TEXTIO 库可用。

8.36 编写一个过程语句，其参数为一个整数信号和一个文件名。文件中的每一行都包含一个延迟值和一个整数。此过程从文件中读取一行文本，并等待长度为延迟值的时间后，把整数值赋给信号，然后再读取另一行文本。当到达文件底部时，过程返回。

8.37 编写一个过程语句，它可以把一个位向量信号的历史值记录到一个文本文件中。每次信号改变时，就在文件中写入当前的时间和信号值。此 VHDL 程序内嵌一个名为 NOW 的函数。此函数被调用后，返回当前仿真时间。



第 9 章　RISC 微处理器设计

微处理器是一种复杂的数字系统。本章中，我们将从 MIPS 技术、MIPS R2000 和 MIPS 处理器的指令集结构（ISA）实现等方面对微处理器进行介绍。指令集结构（ISA）是指机器语言程序员所看到的一些指令、寄存器个数、寻址方式和特定操作时所使用的操作码。我们先简单介绍 RISC 原理，接着再介绍 MIPS ISA。我们对 MIPS 的算术、存储接入和控制转移指令加以讨论。我们将设计实现 ISA 的一个子集，再给出可综合的 MIPS 的一个 VHDL 模块。最后介绍如何使用一个测试平台验证该设计。

9.1 RISC

很多早期的微处理器（如 Intel 8086 和 Motorola 68000 等）都具有一系列功能强的指令集和寻址方式。这就导致了设计的复杂性，尤其是控制单元的复杂性。这些微处理器都包含一个微程序控制单元，这是因为若采用硬线化控制单元，则很难对这样一个复杂的数字系统进行设计和调试（见第 5 章中对微程序和硬线化的比较）。

在 20 世纪 70 年代末和 80 年代初，对微处理器的简化显得非常重要，最终出现了 RISC（精简指令集计算）思想。RISC 处理器是微处理器的一种，它使用一个简易的指令集，而不是使用复杂指令和通用寻址方式。70 年代末到 80 年代初，IBM 公司、斯坦福大学和加利福尼亚大学伯克利分校分别推出最早的 RISC 微处理器，如 IBM801, Stanford MIPS, Berkeley RISC1 和 Berkeley RISC2，它们的设计思路大体相同，都被认为是 RISC 微处理器。与此对应，早期的处理器，如 Intel 8086 和 Motorola 68000/68020 等，都开始被称为复杂指令集（Complex Instruction Set Computing, CISC）处理器。第一代 RISC 处理器包括 MIPS 公司的 MIPS R2000、Sun Microsystems 公司的 SPARC、IBM 公司的 RS/6000 等。IBM 公司的 RS/6000 演变为 POWERPC 和 POWER 结构。

MIPS

在 20 世纪 80 年代初期 MIPS 技术公司是一家生产和销售 RISC 微处理器的公司，它们的第一个 RISC 微处理器为 MIPS R2000。对于计算机设计者来说，MIPS 这个词是指每秒处理的百万级的机器语言指令数，是衡量性能的指标。但是 MIPS 公司名字中的 MIPS 并不是这个意思，它是“Microprocessor without hardware Interlock Processing System”的缩写。在流水线处理器中必须存在一个机制使指令之间相互依赖，所以当一个指令需要前一个指令的结果时，后一个指令不应该进行。这种指令间的依赖性通常通过硬件互锁实现。然而，第一个 MIPS 处理器中没有硬件互锁，这体现了早期 RISC 的思想：能在软件中完成的工作绝不交给硬件来完成。它的流水线互锁是由软件实现的，通过在指令中插入适当数量的 nop（无操作）语句。

大多数 RISC 处理器具有如下特点。

- 统一的指令长度：所有的指令长度相同（32 比特）。这与之前的微处理器有着根本的不同，

之前微处理器的指令长度小可以为 1 字节，大可以到 16 字节。

- 较少的指令格式：RISC 的 ISA 强调指令格式种类越少越好，指令编码的不同区域越规整越好。此特性大大简化了指令译码过程。
- 较少的寻址方式：大多数 RISC 处理器都只支持 1 或 2 个存储寻址方式。不同的寻址方式通过不同方法为指令访问存储器提供地址。比如，有直接寻址、立即寻址、基址加偏值寻址、基址变址寻址和间接寻址。许多 RISC 处理器都仅支持一个寻址方式，一般用寄存器和偏置来给出地址。
- 寄存器数量大：为了防止由于频繁访问内存而造成的性能损失，RISC 通常由大量的寄存器构成。RISC 处理器通常也被称为寄存器-寄存器型结构。所有的算术操作都作用于寄存器操作数。CISC 通常含有 8～12 个寄存器，但是大多数 RISC 却含有 32 个寄存器。
- 加载/存储结构：RISC 的结构通常还被称为加载/存储结构。其核心思想没有直接操作于存储器操作数的算术指令（即直接作用于一个或多个存储器操作数的算术指令）。允许访问内存的指令只有加载和存储指令。加载指令把数据载入到寄存器中，在寄存器中完成算术操作。由于计算时使用的输入和输出操作数都是在寄存器中进行操作的，所以这种结构也被称为寄存器-寄存器型结构。加载/存储结构本身就意味着它也是一种寄存器-寄存器型结构。
- 无隐操作数或副作用：大多数早期的 ISA 都使用隐操作数（如累加器）或隐结果（即副作用），如标记位（条件编码），用于标示条件（如进位、溢出或负号等）。隐操作数和副作用会给流水线和并行执行造成困难。RISC 结构的一个基本原则是尽量少用隐操作数和副作用。

RISC 思想不仅具有以上特点，而且支持设计的简化。术语 RISC 和 CISC 经常作为反义词使用，但是目前还不清楚精简多少和复杂的界限，甚至 RISC 含有的指令集是否比早期的 CISC 处理器少也不是很明确。有些 RISC 的 ISA 含有 100 多个指令，但是有些 CISC 处理器才含有 80 个指令。但是 CISC 中的这 80 个指令要采用多种寻址方式。CISC 处理器 Motorola 68020 支持 20 种不同的寻址方式。把所有格式的指令都考虑在内，与 CISC ISA 相比，大多数 RISC ISA 确实含有较少的指令。RISC 的主要思想是简化：只使用简单的运算符、简化指令格式、减少寻址方式数量、去除复杂操作等。如此的计算通常称为简单指令集计算（Simple Instruction Set Computing, SISC），但是 SISC 听起来与 CISC 很像。

CISC 也不是没有优点。CISC 指令编码比 RISC 指令编码密度大。RISC 的指令长度是固定的，所以对于某些指令来说，使用的空间比实际需要的要多。对于 CISC ISA 来说，它的每个指令长度都与其需要的长度相同，因此其码长较短。在某些环境下（如嵌入式环境），需要指令存储空间小，这时 CISC ISA 就可以显现出优点。

大多数现代微处理器都使用 RISC ISA，如 MIPS R14000, Sun UltraSPARC, IBM PowerPC 和 HP PA-RISC。但是，现代微处理器也有使用 CISC 的，如 Pentium 4 和 x86（x86 处理器是指使用 Intel 8086 及其衍生指令集的各种微处理器，包括 Intel 8086, 80286, 80386, 80486, Pentium, AMD K5, K6, Opteron 等）。

RISC 和 CISC 到底哪个比较好呢？这个问题在 20 世纪 80 年代—90 年代引发了一场深入的争论。现在普遍认为：RISC ISA 在译码和处理方面比较简单，但是硬件具有把复杂 CISC 型指令转化为 RISC 型指令的能力。当今 Pentium 4 和其他高端 x86 处理器都使用 CISC ISA，但是它们都使用硬件把 CISC 指令转化为一个或多个 RISC 型指令，或者转化为更容易实现流水线的微运算符（称为 uops 或 R-ops）。从上面的讨论可以看出，RISC 的简化作用是毋庸置疑的。

MIPS 指令集构架是最早的 RISC ISA 中最简单的一种。它只有一种存储器寻址方式，而其他早期的 RISC，如 SPARC 都含有两种存储器寻址方式。关于 MIPS ISA 的详细内容见参考文献[26]和参考文献[37]。这里我们对 MIPS ISA 只是进行简单介绍。

单指令计算机

过去曾经证明可以使用单指令构建微处理器。这一单指令必须可以访问存储器操作数，可以做算术运算，可以控制转移。减法指令可以作用于存储器操作数，并把结果写入存储器。如果结果是负数且可以被用来编写其他程序，减法指令还可以把这个结果连接到存储器的另一个地址。这样的一个单指令微处理器可以称为 RISC 吗？恐怕不行。虽然它是一个单指令计算机，但是它不是寄存器-寄存器型结构的，而且它的 ISA 只支持简单的操作。我们可以把这种单指令计算机归为 CISC 类中，因为每个指令都是复杂转移并可以访问存储器。关于单指令计算机的详细讨论和实现程序见参考文献[37]。

9.2 MIPS ISA

MIPS ISA 包含简单算术指令、逻辑指令、存储器访问指令、转移指令和跳转指令。MIPS ISA 强调简化性，所以不采用任何比常用指令长的指令。

在 MIPS 内部有 32 个通用寄存器。每个寄存器都是 32 位的。我们使用参考文献[37]中的表示方法标示寄存器，因此 32 个寄存器文件标示为$0, $1, $2, …, $31。对于 ALU 指令，MIPS 采用三地址格式，两个为源地址，另一个为目标地址。例如，把寄存器$3 和$4 加起来并把结果存在$5 中，其指令为

```
add $5, $3, $4
```

下面我们对指令分组介绍。

9.2.1 算术指令

MIPS ISA 中包含计算整数的加法、减法、乘法和除法指令。所有算术指令都总汇于表 9.1 中。有符号数和无符号数的加法和减法可以使用 *add, addu, sub* 和 *subu* 指令实现。有符号算术指令要检测是否溢出，而无符号算术指令无须检测是否溢出。例如，指令

```
sub $5, $3, $4
```

所进行的操作为：用寄存器$3 中的数据减去寄存器$4 中的数据，并把结果存在寄存器$5 中。此指令是有符号指令，所以要检测是否有溢出。

表 9.1 MIPS ISA 中的算术指令

指令	汇编代码		操作	补充说明
add	add	$s1, $s2, $s3	$s1 = $s2 + $s3	检测是否向上溢出
subtract	sub	$s1, $s2, $s3	$s1 = $s2 − $s3	检测是否向上溢出
add immediate	addi	$s1, $s2, k	$s1 = $s2 + k	k 是 16 位常数，符号位扩展后相加；检测是否存在二进制补码向上溢出
add unsigned	addu	$s1, $s2, $s3	$s1 = $s2 + $s3	不检测是否向上溢出
subtract unsigned	subu	$s1, $s2, $s3	$s1 = $s2 − $s3	不检测是否向上溢出
add immediate unsigned	addiu	$s1, $s2, k	$s1 = $s2 + k	不检测是否向上溢出，其余与 addi 相同

（续表）

指　　令	汇编代码		操　　作	补充说明
move from co-processor register	mfc0	$s1, $epc	$s1 = $epc	epc 为特例程序计数器
multiply	Mult	$s2, $s3	Hi, Lo = $s2 × $s3	Hi, Lo 是积，为 64 位有符号数
multiply unsigned	Multu	$s2, $s3	Hi, Lo = $s2 × $s3	Hi, Lo 是积，为 64 位无符号数
divide	Div	$s2, $s3	Lo = $s2 / $s3 Hi = $s2 mod $s3	Lo = 商，Hi = 余数
divide unsigned	Divu	$s2, $s3	Lo = $s2 / $s3 Hi = $s2 mod $s3	商和余数均为无符号数
move from Hi	Mfhi	$s1	$s1 = Hi	把 Hi 复制到$s1
move from Lo	Mflo	$s1	$s1 = Lo	把 Lo 复制到$s1

当检测到有溢出时，就作为例外处理。造成例外的指令地址被保存，控制转移到操作系统来处理这一例外。

如果把指令中一个寄存器改为立即数，那么可以用 *addi* 和 *addiu* 指令完成加法操作。例如，

```
addi $5, $3, 400
```

这个指令把寄存器$3 中的数据与立即数 400 做加运算，并把结果存在寄存器$5 中。在进行加法运算前，立即数必须进行符号扩展。*addiu* 的操作与此类同，只是 *addiu* 指令不会造成溢出意外。

两个 32 位数相乘会得到一个 64 位数，此结果在一个 MIPS 寄存器中无法存储。因此，MIPS 处理器使用两个特殊寄存器 Hi 和 Lo 保存结果。对隐式 Hi 和 Lo 寄存器的使用当然也源于 RISC 的思想。表 9.1 展示了 MIPS ISA 中乘法和除法指令，及何时使用 Hi 和 Lo 寄存器。由于对特殊寄存器的使用，所以我们也需要特殊的指令把数据从这些寄存器传输到目标寄存器中。指令 *mfhi* 和 *mflo* 可以实现此功能。

9.2.2 逻辑指令

MIPS ISA 中的逻辑指令见表 9.2。逻辑指令用来实现寄存器内容的位与和位或操作。当两个操作数都在寄存器中时，用 *and* 和 *or* 指令；当一个操作数在寄存器中，另一个操作数为立即数时，使用 *andi* 和 *ori* 指令；*sll* 和 *srl* 指令实现寄存器内操作数逻辑左移和右移，所移的位数在指令中用立即数表示。

表 9.2 MIPS ISA 中的逻辑指令

指　　令	汇编代码		操　　作	补充说明
And	and	$s1, $s2, $s3	$s1 = $s2 AND $s3	逻辑 AND
Or	or	$s1, $s2, $s3	$s1 = $s2 OR $s3	逻辑 OR
And immediate	andi	$s1, $s2, k	$s1 = $s2 AND k	k 为 16 位常数； k 首先进行 0 扩展
or immediate	ori	$s1, $s2, k	$s1 = $s2 OR k	k 为 16 位常数； k 首先进行 0 扩展
shift left logical	sll	$s1, $s2, k	$s1 = $s2 << k	左移 k 位，k 为 5 位常数
shift right logical	srl	$s1, $s2, k	$s1 = $s2 >> k	左移 k 位，k 为 5 位常数

9.2.3 存储器访问指令

在 MIPS ISA 中只有载入和存储指令可以访问内存。载入指令把数据从内存传输到指定寄存器中。存储指令把寄存器中的数据传输到指定的内存地址。

若使用高级语言进行编程（如 C 语言），需要多少种寻址方式就可以满足效率要求呢？RISC 开发人员经研究确定，只使用基址寄存器和偏值的寻址方式就可以满足效率要求。因为支持存储指令的寻址方式必须由一个基址寄存器和一个有符号偏置构成。内存地址通过计算寄存器中内容与偏置的和可以得到。

观察下面的 MIPS 载入指令：

```
lw $5, 100($4)
```

这个指令通过计算寄存器$4 中内容和偏置 100 的和得到内存地址。当寄存器$4 中的内容为 4000 时，有效地址为 4100，此时地址为 4100 的内存单元中的数据就移到寄存器$5 中。若指令为 sw $6, 100($8)，则可以通过计算寄存器$8 中内容和偏置 100 的和得到有效内存地址，并把寄存器$6 的内容写入指定内存单元中。

在 MIPS 中，每 32 比特称为一个字。MIPS 的指令可以载入和存储字、半字（16 比特）和波特（8 比特）。指令总结见表 9.3。

表 9.3 MIPS ISA 中的内存访问指令

指令	汇编代码		操作	补充说明
load word	lw	$s1, k($s2)	$s1 = Memory[$s2 + k]	从内存中读取 32 位；内存地址 = 寄存器内容+k；k 为 16 为偏置
store word	sw	$s1, k($s2)	Memory[$s2 + k] = $s1	向内存中写入 32 位；内存地址 = 寄存器内容+k；k 为 16 为偏置
load halfword	lh	$s1, k($s2)	$s1 = Memory[$s2 + k]	从内存中读取 16 位；符号扩展并载入寄存器
store halfword	sw	$s1, k($s2)	Memory[$s2 + k] = $s1	向内存中写入 16 位
load byte	lb	$s1, k($s2)	$s1 = Memory[$s2 + k]	从内存中读取一个字节；符号扩展并载入寄存器
store byte	sb	$s1, k($s2)	Memory[$s2 + k] = $s1	向内存中写入一个字节
load byte unsigned	lbu	$s1, k($s2)	$s1 = Memory[$s2 + k]	从内存中读取一个字节；并进行 0 扩展
load upper immediate	lui	$s1, k	$s1 = k*2^{16}	把常数 k 载入到寄存器的高 16 位

9.2.4 控制转移指令

通常程序是按顺序执行的，但是循环、过程和函数语句使程序的控制流程发生改变。当需要改变控制流程时，微处理器使用转移和跳转指令实现控制转移。MIPS ISA 有两种条件转移指令：相等时转移（beq）和不等时转移（bne），如表 9.4 所示。

表 9.4 MIPS ISA 中条件控制相关指令

指令	汇编代码	操作	补充说明
branch on equal	beq $s1, $s2, k	If ($s1 = = $s2) go to PC + 4 + k*4	如果两寄存器相等就转移；PC 转移；目标地址 =PC+4+ 偏置*4；k 为符号扩展

（续表）

指　　令	汇编代码		操　　作	补充说明
branch on not equal	bne	$s1, $s2, k	If ($s1 / = $s2) go to PC + 4 + k*4	如果两寄存器不相等就转移；PC 转移；目标地址= PC + 4 + 偏置*4；k 为符号扩展
set on less than	slt	$s1, $s2, $s3	If ($s2 < $s3) $s1 = 1; else $s1 = 0;	比较并置数（二进制补码）
set on less than immediate	slti	$s1, $s2, k	If ($s2 < k) $s1 = 1; else $s1 = 0;	比较并置数；k 为 16 位常数；符号扩展后进行比较
set on less than unsigned	sltu	$s1, $s2, $s3	If ($s2 < $s3) $s1 = 1; else $s1 = 0;	比较并置数；自然数
set on less than immediate unsigned	sltiu	$s1, $s2, k	If ($s2 < k) $s1 = 1; else $s1 = 0;	比较并置数；k 为 16 位常数；要进行符号扩展；无向上溢出

例如，

```
beq $5, $4, 25
```

这条指令将比较$5 和$4 的内容，如果相等，就转移到指针(pc)(pc)+ 4 + 100。转移指令中固定偏置的值取决于当前程序计算器中指令的个数。MIPS 使用字节为单位进行寻址，因此当转换为字节时，以字为单位的偏置要乘以 4。程序计数器指针位置为下一条指令位置，即当前指针(pc) + 4。因此目标地址为当前指针(pc) + 4 + 4*偏置。偏置是 16 位的，其中有一位是符号位。所以转移的长度仅为+/–32K。

MIPS 只有两个条件转移指令，而 CISC 处理器的条件转移指令包括小于时转移、有进位时转移、有溢出时转移和为负时转移等。MIPS 只包含两个必要的条件转移指令，所以在需要其他条件转移指令时，我们可以把这两条指令结合起来使用。为了判定是否小于或大于，MIPS ISA 提供 *slt* 指令。在进行比较时，这些比较指令根据比较的结果把目标寄存器置 1 或 0。指令 *slt* 通常与指令 *bne* 或 *beq* 结合使用，构成小于时转移和大于时转移等指令。这些指令用于高级编程语言的 *loop* 和 *if-then-else* 语句中。

MIPS ISA 还包含 3 条无条件跳转指令，如表 9.5 所示。它们用于实现函数和过程的调用和返回。

表 9.5　MIPS ISA 中的无条件控制转移指令

指　　令	汇编代码		操　　作	补充说明
jump	j	addr	Go to addr*4; i.e., PC = addr*4	目标地址 = 立即偏置*4；addr 为 26 位
jump register	jr	$reg	Go to $reg; i.e., PC = $reg	$reg 中包含 32 位目标地址
jump and link	jal	addr	return address = PC + 4; go to addr*4	用于过程调用，返回地址并保存在连接寄存器$31 中

跳转指令可以使指针转移到指定的地址。由于 MIPS 指令的长度为 32 位，所以可以用于地址编码的位数为（32–操作码的位数）。在 MIPS 中，操作码占 6 位，因此在跳转指令中只有 26 位可以进行地址编码。为了提高可以进行转移的地址范围，MIPS 设计者指定地址以字为单位，而不是以字节为单位，把得到的地址乘以 4 就可以得到字节形式的地址。

指令 *jr*（寄存器跳转）是间接指令。与之相对比，上一段中提到的跳转指令称为直接跳转指令，因为其指定跳转的地址是直接可见的。在寄存器跳转指令中，我们把寄存器的内容作为要跳转的地址。这种转移指令在 case 语句中经常使用。

跳转和链接指令是专门为过程调用设计的。通过计算指令中指定的偏置，我们可以得到目标地址，但是除了转移到目标地址之外，此指令还要在链接寄存器$31 中保存返回地址。返回地址是指当过程调用结束时，控制指针要返回的地址。返回地址为当前指针(PC) + 4。由于每条指令均为 4 字节长，所以 PC + 4 使指针指向当前指令(*jal* 指令)的下一条指令。

我们已经对 MIPS ISA 中的主要指令类型进行了讨论，为了加强指令的使用熟练度，下面我们介绍一个用汇编语言进行编程的例子。

例 用 MIPS 汇编语言实现下列操作：把两个数组 x(i)和 y(i)相加，每个数组均有 100 个元素。

```
for i = 1, 100, i++          ; 重复 100 次
y(i) = x(i) + y(i)           ; 把两数组中第 i 位数相加
```

假设数组 x 和 y 分别从地址 4000 和 8000 开始存储。

解：

```
          andi   $3 , $3, 0          ; 循环寄存器$3 初始化为 0
          andi   $2 , $2, 0          ; 循环次数寄存器清零
          andi   $2 , $2, 400        ; 循环次数
$label:   lw   $15, 4000($3)         ; 把 x(i)载入到 R15
          lw   $14, 8000($3)         ; 把 y(i)载入到 R14
          add  $24, $15, $14         ; x(i) + y(i)
          sw   $24, 8000($3)         ; 保存新 y(i)
          addi  $3 , $3, 4           ; 更新地址寄存器，地址 = 地址 +4
          bne    $3 , $2, $label     ; 检测是否循环计数器 = 循环次数
```

从 20 世纪 80 年代 MIPS R2000 问世以来，多个微处理器使用了 MIPS ISA。那时候主处理器还没有集成浮点数处理单元。因此浮点数单元作为数学协处理器来实现，即 MIPS R2010。现在，浮点数单元已经集成在主 CPU 内。在 MIPS R2000 处理器之后，MIPS 公司又推出了 MIPS R3000, R4000, R8000, R10000, R12000 和 R14000。这些处理器都使用 MIPS ISA，但是不同的实现采用了不同级别的流水线和不同技术，以得到了高性能。

9.3 MIPS 指令编码

坚持 RISC 的宗旨，MIPS 处理器中所有指令都具有相同的长度：32 位。为了结构简单，MIPS 指令只有三种格式，分别称为 R 格式、I 格式和 J 格式，如表 9.6 所示。

表 9.6 MIPS ISA 中的指令格式

格 式	区 域						使 用
	6 位 31-36	5 位 25-21	5 位 20-16	5 位 15-11	5 位 10-6	6 位 5-0	
R 格式	操作码	rs	rt	rd	shamt	F_code	ALU 指令除了立即指令和跳转寄存器指令
I 格式	操作码	rs	rt	偏置/立即数			载入，存储，立即 ALU 指令，beq, bne
J 格式	操作码	目标地址					跳转(J)，跳转链接(JAL)

R 格式指令主要是 ALU 指令，需要 3 个操作数，其中两个为源操作数（输入寄存器），一个为目标地址（结果寄存器）。寄存器跳转指令(*jr*)也使用此格式。*jr* 指令由 6 个域组成，第一个域为 6 位操作码；接着为 *rs*，*rt* 和 *rd* 三个区，每个域均为 5 位，*rs* 和 *rt* 是源寄存器域，*rd* 是目标寄存器域；下一个域为移位量(shamt)域，这一域指出移位的位数可以为 0 ~ 31 之间任意一个数；最后一个域为附加操作码域，称为函数域 *funct* 或 *F_code*。第一个操作码域只可以进行 2^6 或 64 位指令编码。为了处理多种载入方式（字节载入、半字载入、字载入、浮点数载入等），MIPS 处理器的指令数量要多于 64 个。因此，只用 6 位来设定一条指令是不够的，我们需要更多的空间。于是，MIPS 设计者使用了这样一种结构：R 格式指令的前 6 位均为 0，同时在附加域（指令最后 6 位）中对指令进行进一步设定。

算术指令、载入/存储指令和转移指令都是 I 格式的。I 格式指令要求三个操作数中两个为寄存器，令一个为立即数。操作码区为 6 位，两个寄存器均为 5 位，剩下的 16 位用来设定立即数。对于 *addi* 指令来说，此立即数为一个操作数；对于载入/存储指令来说，立即数为偏置；对于条件转移指令来说，立即数为转移偏置。

J 格式适用于跳转指令。指令字的前 6 位是操作码，其他 26 位用于设定跳转偏置。由于跳转偏置是字地址而不是字节地址，所以此偏置要先乘以 4，然后作为指针 PC 的高 4 位，进而构成一个 32 位的目标地址。MIPS 使用字节寻址访问指令和数据。

MIPS 指令编码详见表 9.7。我们尽可能地使用指令格式中相同的域存储操作码、源寄存器和目标寄存器。例如，在三种指令格式中，我们均使用头 6 位(26 ~ 31)表示操作码，源寄存器和目标寄存器也存储在大体相同的域(21 ~ 25, 16 ~ 20, 11 ~ 15)。这样做在译码时比较简便。

指令编码是很规则的，但是为了使所有的指令都具有相同的长度，必须要做一些必要的妥协。例如，在三寄存器和二寄存器指令格式中，目标寄存器在不同的区域。类似地，在载入指令中，第二个寄存器域是目标寄存器，而在存储指令中，第二个寄存器域存储的是源数据。即使存在一些不规则的地方，但是我们认为指令编码大体上是规则的。

为了更加进一步掌握 MIPS 指令编码，下面我们做几个练习。

表 9.7 MIPS 指令译码

名 称	格 式	区 域						指 令（操作 dest, stc1, str2）
		位 31~26	位 25~21	位 20~16	位 15~11	位 10~6	位 5~0	
add	R	0	2	3	1	0	32	add $1, $2, $3
sub	R	0	2	3	1	0	34	sub $1, $2, $3
addi	I	8	2	1	100			addi $1, $2, 100
addu	R	0	2	3	1	0	33	addiu $1, $2, $3
subu	R	0	2	3	1	0	35	subu $1, $2, $3
addiu	I	9	2	1	100			addiu $1, $2, 100
mfc0	R	16	0	1	14	0	0	mfc0 $1, $epc
mult	R	0	2	3	0	0	24	mult $2, $3
multu	R	0	2	3	0	0	25	multu $2, $3
div	R	0	2	3	0	0	26	div $2, $3
divu	R	0	2	3	0	0	27	divu $2, $3
mfhi	R	0	0	0	1	0	16	mfhi $1

（续表）

名称	格式	区域						指令（操作 dest, stc1, str2）
		位 31～36	位 25～21	位 20～16	位 15～11	位 10～6	位 5～0	
mflo	R	0	0	0	1	0	18	mflo $1
and	R	0	2	3	1	0	36	and $1, $2, $3
or	R	0	2	3	1	0	37	or $1, $2, $3
andi	I	12	2	1	100			andi $1, $2, 100
ori	I	13	2	1	100			ori $1, $2, 100
sll	R	0	0	2	1	10	0	sll $1, $2, 10
srl	R	0	0	2	1	10	2	srl $1, $2, 10
lw	I	35	2	1	100			lw $1, 100($2)
sw	I	43	2	1	100			sw $1, 100($2)
lui	I	15	0	1	100			lui $1, 100
beq	I	4	1	2	25			beq $1, $2, 25
bne	I	5	1	2	25			bne $1, $2, 25
slt	R	0	2	3	1	0	42	slt $1, $2, $3
slti	I	10	2	1	100			slti $1, $2, 100
sltu	R	0	2	3	1	0	43	sltu $1, $2, $3
sltiu	I	11	2	1	100			sltiu $1, $2, 100
j	J	2	2500					j 2500
jr	R	0	31	0	0	0	8	jr $31
jal	J	3	2500					jal 2500

例 写出下列汇编程序的机器编码。

```
        andi $3 , $3, 0           ; 循环寄存器$3 初始化为 0
        andi $2 , $2, 0           ; 循环次数寄存器清零
        andi $2 , $2, 4000        ; 循环次数
$label:    lw  $15, 4000($3)      ; 把 x(i)载入到 R15
        lw  $14, 8000($3)         ; 把 y(i)载入到 R14
        add  $24, $15, $14        ; x(i) + y(i)
        sw  $24, 8000($3)         ; 保存新 y(i)
        addi  $3 , $3, 4          ; 更新地址寄存器，地址 = 地址 +4
        bne   $3 , $2, $label     ; 检测是否循环计数器 = 循环次数
```

解： 以第一条指令 andi $3, $3, 0 为例，我们介绍如何把汇编程序翻译成机器码。

从表 9.7 中我们可以得到 andi 的操作码为 12。因此，第一条指令的头 6 位是 001100，如表 9.8 中第一行最左边所示。接着是源寄存器域，由于源寄存器为$3，所以其内容为 00011。同理，由于目标寄存器为$3，所以目标寄存器域内容为 00011。由于立即数为 0，所以 0～15 位上均为 0。如果用十六进制数表示，该指令的机器码可以表示为 3063 0000。

下面我们再翻译一下最后一条指令：bne $3, $2, label。操作码为 5（即 000101）。下一个域对应$3，所以为 00011。再下一个域对应$2，为 00010。字节偏置为–24，但是指令中的偏置是字格

式的，所以应除以 4，即指令中偏置为–6。用二进制补码表示为 1010，其他位用符号扩展补全，所以 0～15 位上数据为 1111111111111010。

所有指令的机器码见表 9.8。

表 9.8　例题中的 MIPS 机器码

指　令		位 31～26	位 25～21	位 20～16	位 15～11	位 10～6	位 5～0	等效的十六进制数
andi	\$3, \$3, 0	001100	00011	00011	00000	00000	000000	3063 0000
andi	\$2, \$2, 0	001100	00010	00010	00000	00000	000000	3042 0000
addi	\$2, \$2, 4000	001000	00010	00010	00001	11110	100000	2042 0FA0
lw	\$15, 4000(\$3)	100011	00011	01111	00001	11110	100000	8C6F 0FA0
lw	\$15, 8000(\$3)	100011	00011	01110	00011	11101	000000	8C6E 1F40
add	\$24, \$15, \$14	000000	01111	01110	11000	00000	100000	01EE C020
sw	\$24, 8000(\$3)	101011	00011	11000	00011	11101	000000	AC78 1F40
addi	\$3, \$3, 4	001000	00011	00011	00000	00000	000100	2063 0004
bne	\$2, \$2, –6	000101	00011	00010	11111	11111	111010	1462 FFFA

9.4　MIPS 指令子集的实现

本节中，我们讨论 MIPS ISA 的一个子集的简单实现。这里，为了介绍方便，我们构造了一个简单的指令集，它可能只是实际指令集的一部分（子集）。此子集大部分重要指令（包括 ALU、内存访问和转移指令）见表 9.9。本节中我们只是介绍这个指令集的最简实现过程。现代微处理器的实现具有很多特征，例如乘法指令实现、无序执行、转移预测和流水线安排等。为了便于理解，这里我们只介绍了微处理器的有序、无流水线实现。习题中将对其他性能较好的实现过程加以讨论。

表 9.9　本节中实现的 MIPS 指令子集

类别	指令
算术指令	加法指令 减法指令 立即数加法指令
逻辑指令	and 指令 or 指令 立即数 and 指令 立即数 or 指令 逻辑左移 逻辑右移
数据转移指令	载入字指令 存储字指令
有条件跳转指令	相等跳转指令 不相等跳转指令 小于则置数
无条件跳转指令	跳转 寄存器跳转

9.4.1　数据通道设计

在设计微处理器时，我们先弄清指令执行时各个操作的顺序，然后描述完成该指令执行的硬件特性。一般，任何微处理器都按以下方式进行工作：

1. 处理器取出一个指令。
2. 对指令进行译码。译码就是进行指令识别。
3. 处理器读取操作数并执行指令。对于 RISC ISA 来说，算术指令的操作数是存储在寄存器中的。存储输入操作数的寄存器称为源寄存器。对于内存访问指令来说，在进行地址计算时用到寄存器。在指令执行后，处理器把执行后得到的结果写入目标寄存器中（特例：存储指令把结果写入内存）。

因此，设计电路中必须包含指令读取单元、指令译码单元、算术逻辑单元（用于执行指令）、寄存器文件（保存操作数）和内存（存储指令和数据）。下面我们对这几个单元进行详细介绍。

指令读取单元

通常，微处理器都包含一个特殊寄存器：程序计数器(PC)。PC 寄存器用于记录当前指令的下一条指令所在的地址。PC 把该地址发送到指令内存（或指令缓存）中，指令内存返回该地址对应的内存中存储的指令。指令读取完毕后，处理器使 PC 下移，并指向下一条指令。指令读取单元的实现框图见图 9.1。

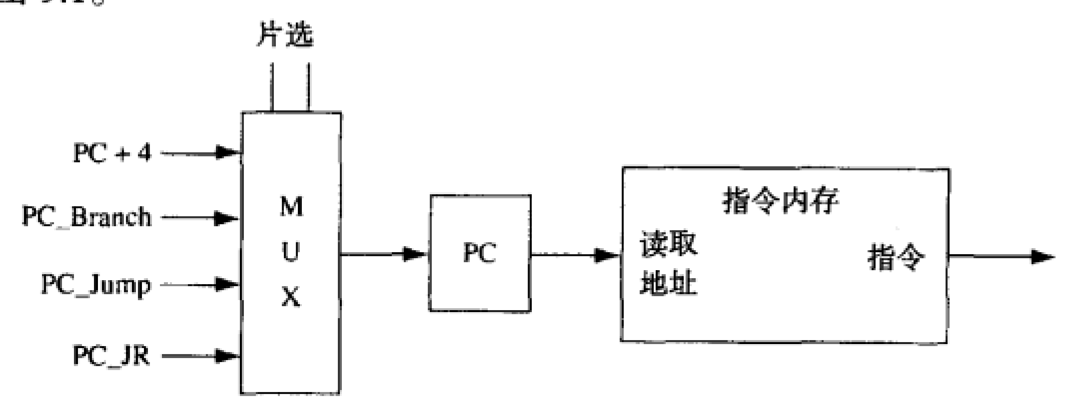


图 9.1 指令抓取框图

根据当前指令的不同，下一个 PC 为以下几种之一。

a. **PC + 4**：由于当前指令为 4 个字节，所以除了转移和跳转指令外，其他所有指令的下一指令地址均为 PC + 4。

b. **PC_Branch**：转移指令（bne 和 beq）中，下一指令的地址可以通过对当前 PC 和指令中指定的偏置求和得到。在 MIPS ISA 中，转移偏置是以字为单位的有符号偏置，第一个字为符号位扩展，可以通过乘以 4 转化为以字节为单位的偏置，求得后再加上当前 PC 的值，则转移指令的下一个 PC 为

$$PC_Branch = PC + 4 + Offset*4$$

c. **PC_Jump**：跳转指令(J)格式中包含目标地址。对于 MIPS ISA 来说，运算符占 32 位中的 6 位，因此最多有 26 位可以作为地址指令编码。为了得到 32 位的跳转地址，首先，我们把指令中 26 位字地址向左移动 2 位变成 28 位字节地址，再把 PC 的高 4 位拼接在最开始，最终构成 32 位地址。这样，跳转指令的下一个 PC 为

$$PC_Jump = PC31..28 \parallel Address*4$$

其中‖表示拼接。

d. **PC_JR:** 寄存器跳转指令(JR)，跳转目标地址是从指令指定的寄存器中得到的。这样，JR 指令的下一个 PC 为

$$PC_JR = [REG]$$

其中[REG]指取寄存器的内容。

目标地址被恰当地计算并馈送回 PC。根据不同指令，我们使用一个多路选择器在转移目标地址、跳转目标地址、寄存器跳转目标地址和 PC + 4 之间选择合适的目标地址。

计算目标地址的时机有多种选择。默认目标地址为 PC + 4，除了 PC 外无须其他任何信息，因此在进行指令读取时就可以直接计算。对于条件转移指令来说，我们可以在读入指令的同时计

算转移的目标地址（PC_Branch）。然而，直到寄存器读入和比较后才能知道是否要进行转移。对于跳转指令来说，由于在指令中就包含了目标地址的信息，所以可以在指令抓取的同时进行目标地址（PC_Jump）的计算。对于寄存器转移指令来说，在寄存器读操作后才可以计算目标地址（PC_JR）。

指令译码单元

由于 RISC ISA 的简易性，其指令译码是非常简单的。从表 9.7 可以看出，MIPS ISA 的指令格式是非常规范化的。大多数情况来说，指令的头 6 位是操作码。但是，由 9.3 节可知，R 格式 ALU 指令的头 6 位是 0，最后 6 位是 **F_code**，用于进一步辨别指令。

操作码用于辨别指令和指令格式。指令格式的规范化使得很多指令区域可以直接用于寄存器寻址和控制信号生成。指令操作码馈送到控制单元，由控制单元生成各种控制信号。

指令执行单元

当指令辨识已经到译码阶段时，下一步就是读取操作数并进行相关操作。在 RISC 指令集中，操作数保存在寄存器中。MIPS 含有 32 个寄存器，我们可以通过寄存器文件找到这些寄存器。寄存器文件至少要有两个读端口以同时支持两个操作数的读操作，并且还要有一个写端口。

寄存器文件按以下步骤进行操作。保存输入操作数的寄存器称为源寄存器，接收结果的寄存器称为目标寄存器。源寄存器的地址记录在寄存器文件中。寄存器文件应该从相应的寄存器中读取数据并放置到输出数据线上，数据被馈送到 ALU 进行计算。ALU 中含有功能单元（如加法器、移位器等），还含有更复杂的单元（如乘法器等）。本章中我们设计的 ALU 中不包含乘法器。

在大多数指令中，ALU 得到的结果应该写入目标寄存器。执行该操作时，ALU 中的结果应放置到寄存器的输入数据线上。目标寄存器名和寄存器写命令被应用于寄存器文件中，最后输入数据写入目标寄存器。

数据通道用于执行 ALU 和内存指令，其实现框图见图 9.2。数据通道中包含一个 ALU，它可以进行如下操作：加、减、与、或。对于 R 格式指令，ALU 的两个操作数都是从寄存器中读入的。对于 I 格式指令，第二个操作数是立即数，其符号位要进行扩展。由于 ALU 的操作数不是从寄存器中得到的，就是从符号扩展器中得到的，所以我们使用多路选择器选择适当的操作数。

非算术指令也需要使用 ALU。对于内存访问指令，我们首先计算将要访问的地址。ALU 便用于进行地址计算。载入和存储指令也需要进行地址计算，它们的第一个操作数是从寄存器中得到的，并通过对指令中立即数偏置进行符号扩展，我们得到第二个操作数。

条件转移指令也需要使用 ALU。MIPS 只有两个条件转移指令：beq 和 bne。我们可以使用 ALU 比较两个寄存器中的数据是否相等。进行比较的两个操作数可以从寄存器文件中得到。数据通道还必须包含数据内存单元。这是因为载入和存储指令对数据内存单元进行访问。现代微处理器还包含一片数据缓存。在这里，我们不会设计数据缓存，但是我们假设存在这样一片数据内存：当把数据地址提供给内存后，指令可以进行内存访问。

总数据分支

总数据分支见图 9.3。它集成了如图 9.1 和图 9.2 所示的抓取和执行硬件，并添加了其他元件以保证操作的正确运行。图 9.3 中还对控制信号进行了说明。

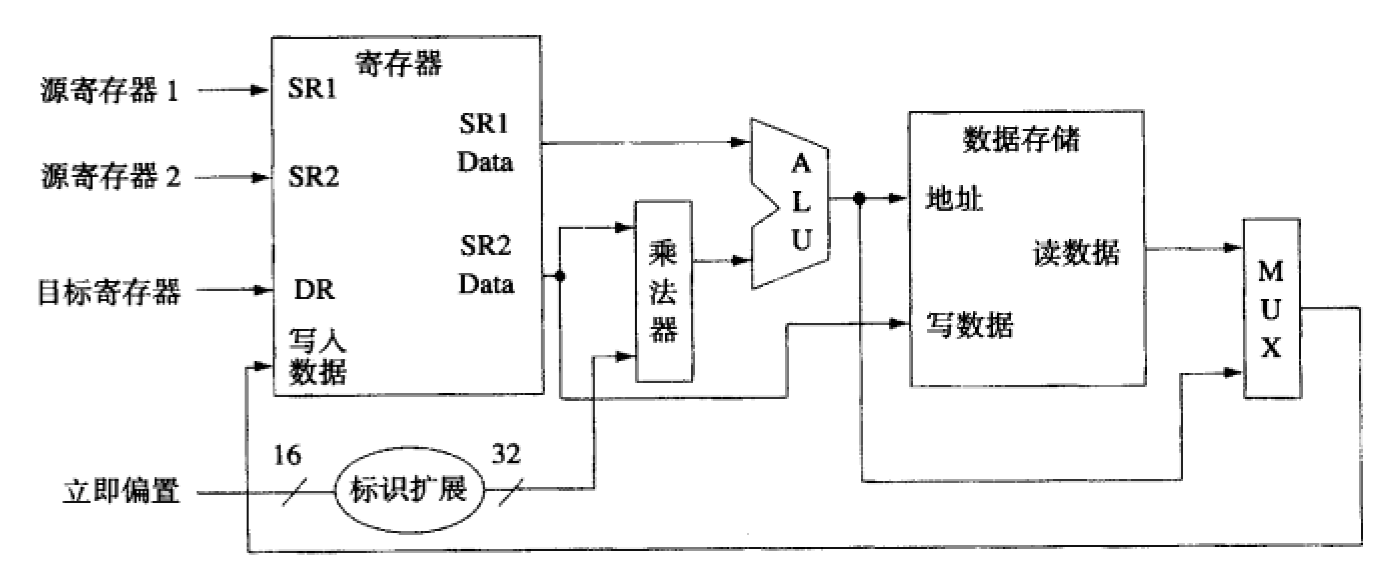


图 9.2 计算和内存指令所需的数据通道



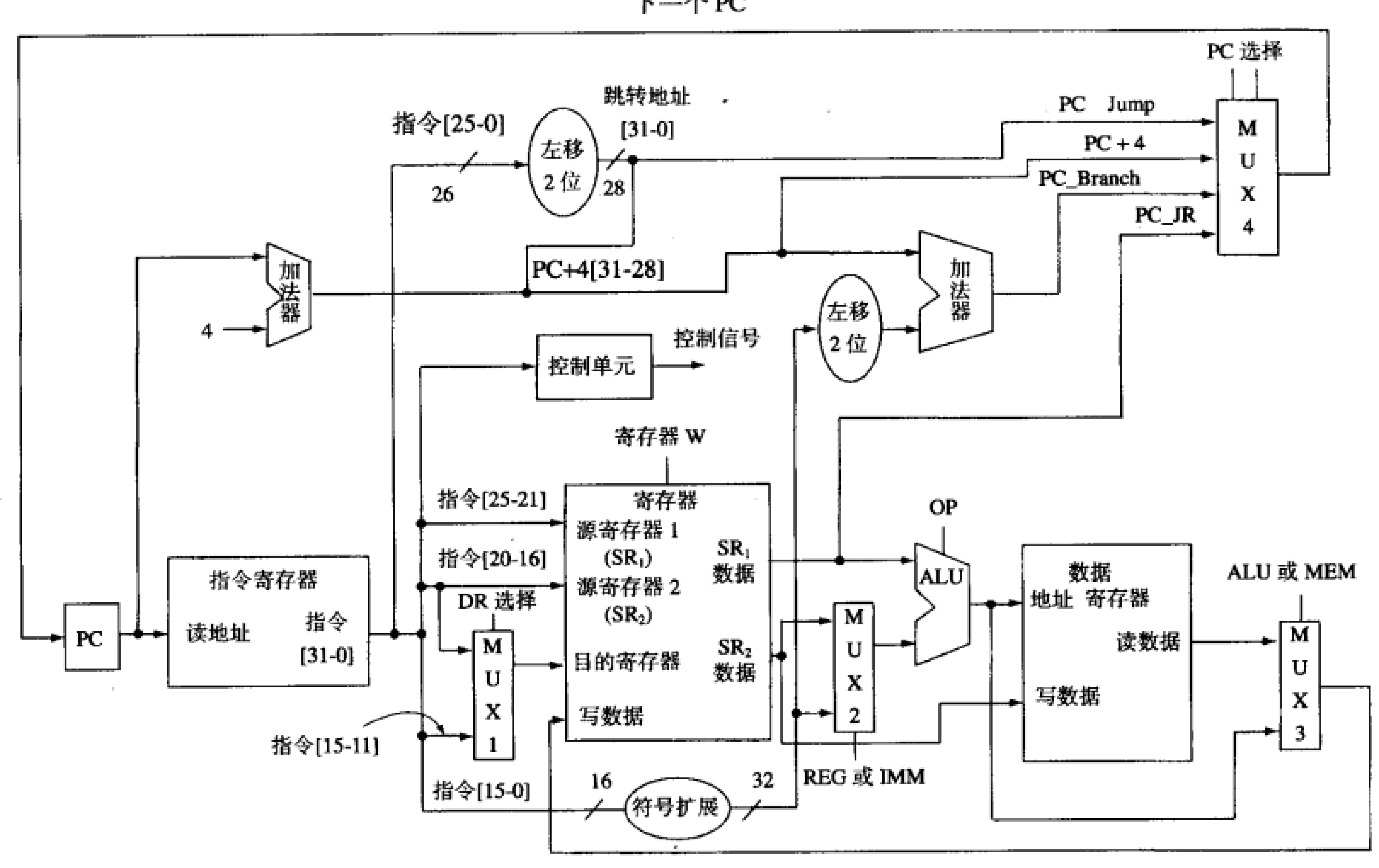


图 9.3 总数据分支

图 9.3 中还展示了如何使用各个多路选择器，以及指令的不同位是怎样与寄存器文件连接的。所有 ALU 指令的 21 ~ 25 位包含一个源寄存器地址（参见表 9.6）。因此 21 ~ 25 位可以直接与寄存器文件中的第一个源寄存器相连。任何含有第二个源寄存器的指令将在 16 ~ 20 位存储第二个源寄存器的地址，因此 16 ~ 20 位可以直接与寄存器文件中的源寄存器相连。但是，对于不同指令来说，其目标寄存器地址存储在不同区域。对于 R 格式指令来说，其目的寄存器地址存储在 11 ~ 15 位；对于 I 格式指令来说，其目标寄存器地址存储在 16 ~ 20 位。因此，我们需要两个多路选择器，一个用于选择恰当的目标寄存器地址，另一个用于选择立即数或寄存器操作数作为 ALU 的操作数。

各种指令目标地址的计算在图 9.3 中也有详细的介绍。PC + 4 的默认下一个地址可以通过加法器进行计算。对 PC 转移偏置的计算也可以通过单独的加法器实现。乘法器也可以用于选择适当的 PC。

总之，

- MUX1 根据指令格式从恰当的寄存器区域中选择目标寄存器地址。R 格式指令的目标寄存器地址在 16 ~ 20 位；I 格式指令的目标寄存器地址在 11 ~ 15 位。
- MUX2 为 ALU 选择第二个操作数，第二个操作数或者从寄存器中得到，或者为立即数。对于 R 格式 ALU 指令和条件转移指令，选择寄存器；对于 I 格式 ALU 指令，选择立即数。
- MUX3 选择内存输出或 ALU 输出，作为数据传输到目标寄存器。对于载入指令，选择内存数据。
- MUX4 根据指令的种类从 4 个可能的下一 PC 中选择一个。

9.4.2 指令执行流程

指令的执行流程示于如图 9.4。

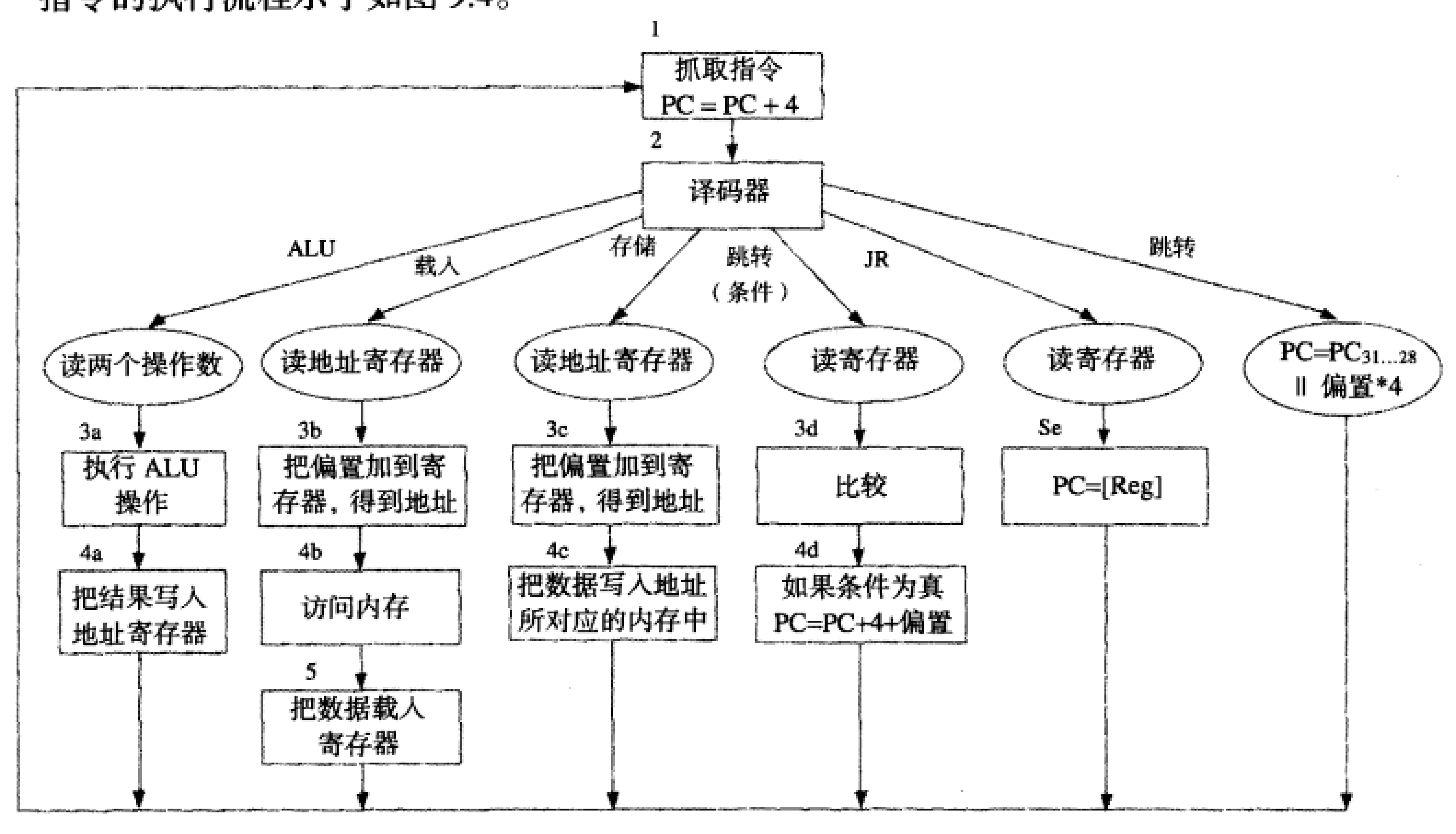


图 9.4 指令处理流程图

对于所有指令，第一步均为指令读取。程序计数器(PC)中的地址送到指令内存单元。所有指令均需更新 PC 指针，使其指向下一条指令。除了转移和跳转指令，大多数指令都是顺序执行的，因此PC的更新只需指向顺序执行的下一条指令即可。转移和跳转指令的PC更新则稍后适当进行。

第二步为译码。根据操作码的不同进行不同操作。对于 R 格式指令和部分 I 格式指令（如 bne 和 beq）来说，ALU 的两个操作数都是从寄存器中读取的。对于其他 I 格式指令来说，一个操作数是从寄存器中读取的，另一个操作数是立即数，还要进行符号位扩展。寄存器跳转指令(jr)是一个 R 型指令，它从源寄存器中读取操作数。每个指令是否需要 ALU 操作也在译码中进行检验。例如，指令 bne 和 beq 需要进行减操作，载入和存储指令需要进行加操作；跳转指令需要计算跳转目标地址。计算完毕后，由于跳转指令无需其他操作，所以其控制流程回到第 1 步。

第三步是指令的实际执行。根据指令要求，在这一步中 ALU 将进行不同操作，且不同类型指令的不同操作详见标有 3a, 3b 等的方框。根据指令的类型，每个指令只选择这些操作中的一种。除了跳转指令，其他指令都必须经过此步骤。虽然寄存器跳转指令(jr)无须进行任何算术操作，但

是第二步中从寄存器中读取的数据必须载入到 PC 中，而且载入和存储指令需要使用 ALU 中的加法操作计算内存地址。

不同指令的第四步操作有所不同。R 型和 I 型算术逻辑指令要把它们的计算结果写入目标寄存器。转移指令必须对它的条件进行检测并确定是否进行转移。如果要进行转移，就需要计算目标地址。载入指令将启动内存读操作。对于内存存储指令来说，从第二个源寄存器中得到的数据写入内存。并进行内存写操作初始化。除了载入操作，对于其他指令来说，此步骤为执行流程的最后一步。

载入指令需要进行第五步。内存输出数据被写入目标寄存器。

我们可以通过不同方法写出指令执行流程。在大多数简单实现中，我们使用很慢的时钟，并且处理器在一个时钟周期内实现每个指令的所有操作。其缺点在于，所有指令的执行时间都与最慢的指令相同，因为时钟周期必须足够长以满足最慢指令的需求。还有一种选择，就是每个指令可以不在一个时钟周期内实现，但是其所需的时钟周期只能刚好完成每类指令的全部操作。例如，图 9.4 可以看做是一个 SM 图，其中每个方框的实现都需要一个时钟周期。这时，跳转指令的实现需要两个时钟周期，ALU 指令的实现需要四个时钟周期，载入指令需要五个时钟周期。在下一节中，我们将给出这一实现 VHDL 模块。

9.5 VHDL 模块

处理器的 VHDL 模块结构图 9.5。在此结构中，指令内存、数据内存和寄存器文件都作为元件出现，它们各有自己的实体-结构体对。主代码为 MIPS，实体部分控制各个不同阶段操作中指令的顺序。为简单起见，我们把指令和数据内存单元结合为一个内存单元，程序中还展示了地址和数据总线的使用。随后，我们要用一个测试平台时，让它直接写入指令内存中，以便测试待测指令。

让我们先对寄存器和内存单元建模。

9.5.1 寄存器文件的 VHDL 模块

寄存器文件的 VHDL 模块示于图 9.6 所示。REG 实体用来表示 32 个 MIPS 寄存器。每个寄存器都是 32 位的。目标寄存器地址为 *DR*，源寄存器地址为 *SR1* 和 *SR2*。由于有 32 个寄存器，所以 *DR*, *SR1* 和 *SR2* 均为 5 位。输出 *ReadReg1* 和 *ReadReg2* 为寄存器 *SR1* 和 *SR2* 内的数据。*ReadReg1* 直接馈回 ALU；*ReadReg2* 作为 ALU 的第二个输入。对于存储指令来说，*ReadReg2* 用做数据内存的输入。控制信号 *RegW* 用来控制寄存器文件的写操作。当 *RegW* 为真时，数据线 *Reg_In* 上的数据就写入 *DR* 指向的寄存器中。

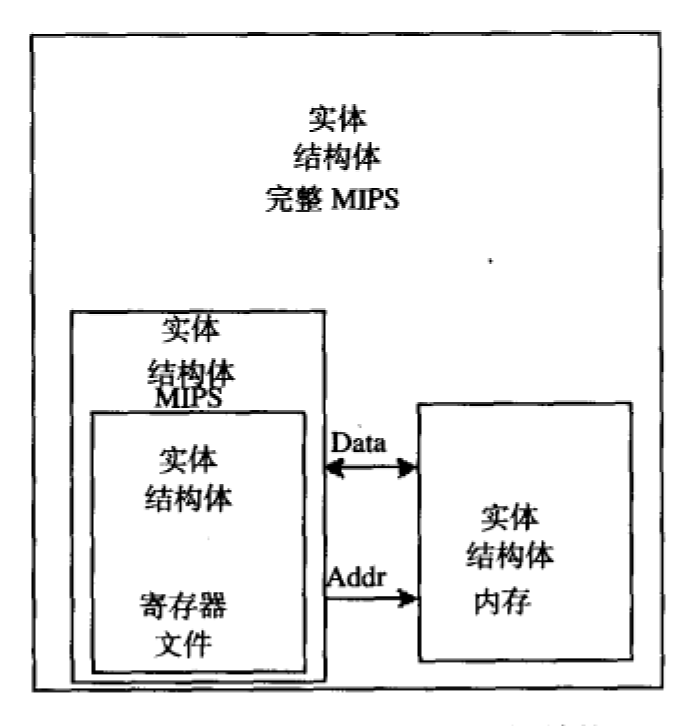


图 9.5 处理器的 VHDL 组织结构

如果该代码使用 Xilinx Spartan FPGA 进行综合，为了使寄存器文件映射到分布式 RAM 中，则程序读操作必须是异步的。我们期望读操作是同步的，但是由于随后寄存器文件要使用当前 Xilinx 综合工具映射到块 RAM 中，所以我们使用异步读操作以满足寄存器文件的分布式 RAM 的生成。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity REG is
port(CLK: in std_logic;
     RegW: in std_logic;
     DR, SR1, SR2: in unsigned(4 downto 0);
     Reg_In: in unsigned(31 downto 0);
     ReadReg1, ReadReg2: out unsigned(31 downto 0));
end REG;

architecture Behavioral of REG is
  type RAM is array (0 to 31) of unsigned(31 downto 0);
  signal Regs: RAM := (others => (others => '1')); -- set all reg bits to '1'
begin
  process(clk)
  begin
    if CLK = '1' and CLK'event then
      if RegW = '1' then
        Regs(to_integer(DR)) <= Reg_In;
      end if;
    end if;
  end process;
  ReadReg1 <= Regs(to_integer(SR1)); -- asynchronous read
  ReadReg2 <= Regs(to_integer(SR2)); -- asynchronous read
end Behavioral;
```

图 9.6　寄存器文件的 VHDL 程序

9.5.2 内存的 VHDL 模块

内存单元的 VHDL 模块示于图 9.7。该 VHDL 代码与第 8 章介绍的 SRAM 模块很相似。该 SRAM 模块具有三态输入输出线，并允许使用测试平台对其进行简单测试。测试时，测试平台把指令写入内存，处理器就可以读取指令并进行数据读/写操作。测试平台和处理器可以驱动内存的数据总线。虽然图 9.3 给出的分开的指令内存和数据内存，使用方便并有利于说明地址总线和数据总线的使用，但我们还是用统一的内存模块存储指令和数据。此内存有 128 个存储单元，每个单元均为 32 位。我们假设这一存储器的前 64 个单元存储指令，后 64 个单元用于存储数据。地址总线为 32 比特宽，但是我们只用低 7 位，因为我们要构建小的内存。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Memory is
  port(CS, WE, Clk: in std_logic;
       ADDR: in unsigned(31 downto 0);
       Mem_Bus: inout unsigned(31 downto 0));
end Memory;

architecture Internal of Memory is
  type RAMtype is array (0 to 127) of unsigned(31 downto 0);
  signal RAM1: RAMtype := (others => (others => '0'));
  signal output: unsigned(31 downto 0);
begin
  Mem_Bus <= (others => 'Z') when CS = '0' or WE = '1'
    else output;
  process(Clk)
  begin
    if Clk = '0' and Clk'event then
      if CS = '1' and WE = '1' then
        RAM1(to_integer(ADDR(6 downto 0))) <= Mem_Bus;
      end if;
    output <= RAM1(to_integer(ADDR(6 downto 0)));
    end if;
  end process;
end Internal;
```

图 9.7　指令/数据通用存储器的 VHDL 代码

处理器适当地驱动地址总线为指令和数据访问服务。地址输入可以从程序计数器 PC 中得到

用以读取指令，也可以从 ALU 中计算得到用以存取内存数据。片选(*CS*)和写允许(*WE*)信号控制处理器能否进行读写操作。当 *CS* 和 *WE* 均为真时，*Mem_Bus* 上的数据写入 *ADDR* 指向的内存单元。

为简单起见，VHDL 代码中内存的地址都是以字为单位的。因此，在图 9.8 中转移和跳转偏置没有乘以 4。在实际 MIPS 处理器中，内存地址是以字节为单位的。因此，当每个指令访问内存时，所得到的数据必须是当前地址指针指向的内存单元及其后三个内存单元中的数据。例如，如果 address = 0，则指令寄存器必须载入 MEM[0], MEM[1], MEM[2]和 MEM[3]。指令的存储依赖于机器中高低字节的优先顺序（见补充说明）。很多现代微处理器既支持高字节优先字节顺序（Big-endian），也支持低字节优先字节顺序（Little-endian）。

9.5.3 处理器 CPU 的 VHDL 代码

本节中，我们介绍微处理器的中央处理单元（CPU）的 VHDL 代码。这里要用前面介绍的寄存器模块。表 9.9 中 MIPS 指令的 VHDL 模块示于图 9.8。该 VHDL 模块按照图 9.4 的流程，顺序实现指令的读取、译码和执行。为了提供代码的可读性，我们定义了多个名称。例如，指令最重要的高 6 位定义为 *Opcode*，低 6 位定义为 *F_Code*，移位指令的移位数定义为 *NumShift*，两个源寄存器定义为 *SR1* 和 *SR2*。下面的语句完成这些定义：

```
alias opcode: unsigned(5 downto 0) is Instr(31 downto 26);
alias SR1: unsigned(4 downto 0) is Instr(25 downto 21);
alias SR2: unsigned(4 downto 0) is Instr(20 downto 16);
alias F_Code: unsigned(5 downto 0) is Instr(5 downto 0);
alias NumShift: unsigned(4 downto 0) is Instr(10 downto 6);
```

低字节优先字节顺序和高字节优先字节顺序

当我们把 16 位或 32 位数据存储以字节为单位的内存中时，可以用两种方式存放数据：低字节优先字节顺序和高字节优先字节顺序。低字节优先字节顺序是先存放最低有效位字节（MSB），而高字节优先字节顺序则是先存放最高有效位字节（LSB）于地址低端。下面，我们举例说明如何使用这两种方法将一条 MIPS 指令存入内存。MIPS 指令为 andi $3，0；可以编码为 30630000（十六进制表示）。假设要将其存入到地址 2000，分别使用低字节优先字节顺序和高字节优先字节顺序两种方法进行存储，则内存单元如下所示：

地址	高字节优先字节顺序	低字节优先字节顺序
2000	30	00
2001	63	00
2002	00	63
2003	00	30

为了提高可读性，我们还使用了常数声明语句，用来表示与表 9.7 所对应的多种操作码。例如，载入指令 *lw* 的操作码为 35，存储指令 *sw* 的操作码为 43，二者操作码的 VHDL 常数声明语句为

```
constant lw  : unsigned(5 downto 0) := "100011";  -- 35
constant sw  : unsigned(5 downto 0) := "101011";  -- 43
```

立即数的符号扩展由下列语句完成：

```
Imm_Ext <= x"FFFF"&Instr(15 downto 0) when Instr(15) = '1' else
           x"0000"&Instr(15 downto 0);
```

VHDL 模块中使用的信号如下所示：

CLK（输入）	时钟信号。
Rst（输入）	同步复位信号。
CS（输出）	内存片选信号。当 CS 有效、WE 无效时，内存模块把 Addr 指向的内存单元的数据输出到 men_bus 总线上。
WE（输出）	内存写允许信号。当 WE 和 CS 都有效时，在时钟下降沿到来时，内存模块把数据存储到 Addr 指向的内存单元。
Addr（输出）	内存地址信号。在状态 0（从内存中读取指令）。Addr 连接 PC。否则，Addr 连接 ALU 的计算结果（32 位）。
Mem_Bus（输入/输出）	三态内存总线。携带内存模块的存取数据。在内存写操作时，MIPS 模块输出到总线；在内存读操作时，内存模块输出到总线。当不被使用时，总线处于高阻态（Hi-Z）。
Op	ALU 操作选择信号。在进行指令译码时决定 ALU 应进行何种操作（如加、与、或等）。
Format	指出当前指令的格式（R 型、I 型或 J 型）
Instr	当前指令（32 位）
Imm_Ext	指令中立即符号扩展的立即数（32 位）
PC	当前程序计数器（32 位）
NPC	下一个程序计数器（32 位）
ReadReg1	第一个源寄存器（SR1）的内容（32 位）
ReadReg2	第二个源寄存器（SR2）的内容（32 位）
Reg_In	寄存器数据输入。当执行载入指令时，Reg_In 与内存总线相连，否则与 ALU 的计算结果相连（32 位）。
ALU_InA	ALU 的第一个操作数（32 位）
ALU_InB	ALU 的第二个操作数。当指令为立即数模式时，ALU_InB 与 Imm_Ext 相连。否则，与 ReadReg2（32 位）相连。
ALU_Result	ALU 输出（32 位）
ALUorMEM	Reg_In 多路选择器选择信号。它指出寄存器输入应该来自内存还是 ALU。
REGorIMM	ALU_InB 多路选择器选择信号。它指出 ALU 的第二个操作数是来自寄存器输出还是符号位扩展后的立即数。
RegW	指出目的寄存器是否应该被写入。因为有些指令不把任何结果写入寄存器（例如转移指令和存储指令）。
FetchDorl	地址多路选择器选择信号。它确定 Addr 指向将要读取的指令，还是指向将要读取或写入的数据。
Writing	使 MIPS 处理器输出到内存总线的控制信号。除了内存写操作外，其他情况下输出均为'Hi-Z'，所以总线可以用于其他模块。注意 Writing 不能用 WE 代替，因为 WE 的端口模式为 OUT，而 Writing 用于端口模式 IN。

```
DR          目标寄存器地址（5 位）
State       当前状态
nState      下一状态
```

此 VHDL 代码中使用了两个进程。由于我们对读取、译码和执行等操作使用另外的时钟周期，所以必须要保存每个阶段生成的信号以备后续使用。保存语句为：

```
OpSave <= Op;
REGorIMM_Save <= REGorIMM;
ALUorMEM_Save <= ALUorMEM;
ALU_Result_Save <= ALU_Result;
```

这些语句在时钟有效进程（第二个进程）中用于保存相关信号。

程序计数器输入处的多路选择器没有明显的编程语句。在各种状态下，数据的各种传输都是用行为描述方式编程的。一个好的综合器可以生成用于实现多种数据传输的多路选择器。类似地，用于选择目标寄存器地址的多路选择器也没有明显的编程语句。如果综合工具对于这种多路数据传输生成低效率的硬件，则我们必须把多路选择器的代码写入数据通道，并生成选择控制信号。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity MIPS is
  port(CLK, RST: in std_logic;
       CS, WE: out std_logic;
       ADDR: out unsigned (31 downto 0);
       Mem_Bus: inout unsigned(31 downto 0));
end MIPS;

architecture structure of MIPS is
  component REG is
    port(CLK: in std_logic;
         RegW: in std_logic;
         DR, SR1, SR2: in unsigned(4 downto 0);
         Reg_In: in unsigned(31 downto 0);
         ReadReg1, ReadReg2: out unsigned(31 downto 0));
  end component;
  type Operation is (and1, or1, add, sub, slt, shr, shl, jr);
  signal Op, OpSave: Operation := and1;
  type Instr_Format is (R, I, J);  -- (Arithmetic, Addr_Imm, Jump)
  signal Format: Instr_Format := R;
  signal Instr, Imm_Ext: unsigned (31 downto 0);
  signal PC, nPC, ReadReg1, ReadReg2, Reg_In: unsigned(31 downto 0);
  signal ALU_InA, ALU_InB, ALU_Result: unsigned(31 downto 0);
  signal ALU_Result_Save: unsigned(31 downto 0);
  signal ALUorMEM, RegW, FetchDorI, Writing, REGorIMM: std_logic := '0';
  signal REGorIMM_Save, ALUorMEM_Save: std_logic := '0';
  signal DR: unsigned(4 downto 0);
  signal State, nState: integer range 0 to 4 := 0;
  constant addi: unsigned(5 downto 0) := "001000"; -- 8
  constant andi: unsigned(5 downto 0) := "001100"; -- 12
  constant ori:  unsigned(5 downto 0) := "001101"; -- 13
  constant lw:   unsigned(5 downto 0) := "100011"; -- 35
```

图 9.8　MIPS 子集 VHDL 代码

```
  constant sw:   unsigned(5 downto 0) := "101011"; -- 43
  constant beq:  unsigned(5 downto 0) := "000100"; -- 4
  constant bne:  unsigned(5 downto 0) := "000101"; -- 5
  constant jump: unsigned(5 downto 0) := "000010"; -- 2
  alias opcode: unsigned(5 downto 0) is Instr(31 downto 26);
  alias SR1: unsigned(4 downto 0) is Instr(25 downto 21);
  alias SR2: unsigned(4 downto 0) is Instr(20 downto 16);
  alias F_Code: unsigned(5 downto 0) is Instr(5 downto 0);
  alias NumShift: unsigned(4 downto 0) is Instr(10 downto 6);
  alias ImmField: unsigned (15 downto 0) is Instr(15 downto 0);
begin
  A1: Reg port map (CLK, RegW, DR, SR1, SR2, Reg_In, ReadReg1, ReadReg2);
  Imm_Ext <= x"FFFF" & Instr(15 downto 0) when Instr(15) = '1'
    else x"0000" & Instr(15 downto 0);  -- Sign extend immediate field
  DR <= Instr(15 downto 11) when Format = R
    else Instr(20 downto 16);           -- Destination Register MUX (MUX1)
  ALU_InA <= ReadReg1;
  ALU_InB <= Imm_Ext when REGorIMM_Save = '1' else ReadReg2; -- ALU MUX (MUX2)
  Reg_in <= Mem_Bus when ALUorMEM_Save = '1' else ALU_Result_Save; -- Data MUX
  Format <= R when Opcode = 0 else J when Opcode = 2 else I;
  Mem_Bus <= ReadReg2 when Writing = '1' else
    "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; -- drive memory bus only during writes
  ADDR <= PC when FetchDorI = '1' else ALU_Result_Save; --ADDR Mux

  process(State, PC, Instr, Format, F_Code, opcode, Op, ALU_InA, ALU_InB, Imm_Ext)
  begin
    FetchDorI <= '0'; CS <= '0'; WE <= '0'; RegW <= '0'; Writing <= '0';
    ALU_Result <= "00000000000000000000000000000000";
    npc <= pc; Op <= jr; REGorIMM <= '0'; ALUorMEM <= '0';
    case state is
      when 0 =>  --fetch instruction
        nPC <= PC + 1; CS <= '1'; nState <= 1; -- increment by 1 since word address
        FetchDorI <= '1';
      when 1 =>
        nState <= 2; REGorIMM <= '0'; ALUorMEM <= '0';
        if Format = J then
          nPC <= "000000" & Instr(25 downto 0); nState <= 0; --jump, and finish
          -- offset not multiplied by 4 since mem is word address
        elsif Format = R then  -- register instructions
          if    F_code = "100000" then Op <= add;  -- add
          elsif F_code = "100010" then Op <= sub;  -- subtract
          elsif F_code = "100100" then Op <= and1; -- and
          elsif F_code = "100101" then Op <= or1;  -- or
          elsif F_code = "101010" then Op <= slt;  -- set on less than
          elsif F_code = "000010" then Op <= shr;  -- shift right
          elsif F_code = "000000" then Op <= shl;  -- shift left
          elsif F_code = "001000" then Op <= jr;   -- jump register
          end if;
        elsif Format = I then -- immediate instructions
          REGorIMM <= '1';
          if Opcode = lw or Opcode = sw or Opcode = addi then Op <= add;
          elsif Opcode = beq or Opcode = bne then Op <= sub; REGorIMM <= '0';
          elsif Opcode = andi then Op <= and1;
          elsif Opcode = ori then Op <= or1;
          end if;
          if Opcode = lw then ALUorMEM <= '1'; end if;
        end if;
      when 2 =>
        nState <= 3;
        if    OpSave = and1 then ALU_Result <= ALU_InA and ALU_InB;
        elsif OpSave = or1 then ALU_Result <= ALU_InA or ALU_InB;
        elsif OpSave = add then ALU_Result <= ALU_InA + ALU_InB;
        elsif OpSave = sub then ALU_Result <= ALU_InA - ALU_InB;
        elsif OpSave = shr then ALU_Result <= ALU_InB srl to_integer(numshift);
        elsif OpSave = shl then ALU_Result <= ALU_InB sll to_integer(numshift);
        elsif OpSave = slt then -- set on less than
          if ALU_InA < ALU_InB then ALU_Result <= X"00000001";
```

图 9.8（续 1）　MIPS 子集 VHDL 代码

```
          else ALU_Result <= X"00000000";
          end if;
        end if;
        if ((ALU_InA = ALU_InB) and Opcode = beq) or
           ((ALU_InA /= ALU_InB) and Opcode = bne) then
          nPC <= PC + Imm_Ext; nState <= 0;
        elsif opcode = bne or opcode = beq then nState <= 0;
        elsif OpSave = jr then nPC <= ALU_InA; nState <= 0;
        end if;
      when 3 =>
        nState <= 0;
        if Format = R or Opcode = addi or Opcode = andi or Opcode = ori then
          RegW <= '1';
        elsif Opcode = sw then CS <= '1'; WE <= '1'; Writing <= '1';
        elsif Opcode = lw then CS <= '1'; nState <= 4;
        end if;
      when 4 =>
        nState <= 0; CS <= '1';
        if Opcode = lw then RegW <= '1'; end if;
    end case;
  end process;

  process(CLK)
  begin
    if CLK = '1' and CLK'event then
      if rst = '1' then
        State <= 0;
        PC <= x"00000000";
      else
        State <= nState;
        PC <= nPC;
      end if;
      if State = 0 then Instr <= Mem_Bus; end if;
      if State = 1 then
        OpSave <= Op;
        REGorIMM_Save <= REGorIMM;
        ALUorMEM_Save <= ALUorMEM;
      end if;
      if State = 2 then ALU_Result_Save <= ALU_Result; end if;
    end if;
  end process;
end structure;
```

图 9.8（续 2） MIPS 子集 VHDL 代码

9.5.4 完整的 MIPS 模块

处理器模块和内存模块集成在一起得到完整的 MIPS 模块（图 9.9）。处理器模块和内存模块用元件说明语句定义元件，用端口映射语句（port-map）引用这些元件。最高层实体名为 Complete_MIPS。在最高层实体中我们把地址和数据总线作为输出引出。如果在实体中没有定义输出，那么在进行综合时就会得到空的模块。对于不同的综合工具，可能把未使用的信号（及对应的连接）从综合电路中去除掉。

我们综合了图 9.9 模块。综合工具为 Xilinx ISE，并在 Spartan 3 FPGA 上实现。该 FPGA 有 1108 个 4 输入 LUT、660 个片、111 个触发器和 1 个块 RAM。寄存器文件使用 194 个 4 输入 LUT。由于 1 个 LUT 可以提供 16 比特存储空间，所以 32 个 32 位寄存器需要 64 个 LUT 的存储空间。由于寄存器文件有两个读端口，所以需要 128 个 LUT。其他的 LUT 用于地址译码和控制信号。为了在原型开发用板上实现此设计，必须附加输入接口和显示模块。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Complete_MIPS is
  port(CLK, RST: in std_logic;
       A_Out, D_Out: out unsigned(31 downto 0));
end Complete_MIPS;

architecture model of Complete_MIPS is
  component MIPS is
    port(CLK, RST: in std_logic;
         CS, WE: out std_logic;
         ADDR: out unsigned(31 downto 0);
         Mem_Bus: inout unsigned(31 downto 0));
  end component;
  component Memory is
    port(CS, WE, Clk: in std_logic;
         ADDR: in unsigned(31 downto 0);
         Mem_Bus: inout unsigned(31 downto 0));
  end component;
  signal CS, WE: std_logic;
  signal ADDR, Mem_Bus: unsigned(31 downto 0);
begin
  CPU: MIPS port map (CLK, RST, CS, WE, ADDR, Mem_Bus);
  MEM: Memory port map (CS, WE, CLK, ADDR, Mem_Bus);
  A_Out <= ADDR;
  D_Out <= Mem_Bus;
end model;
```

图 9.9　集成处理器和内存单元的 VHDL 程序

9.5.5　处理器模块测试

整个 MIPS VHDL 模块用图 9.10 的测试平台进行测试，测试平台必须验证每个指令是否操作正确。该测试平台由测试指令的 MIPS 程序和把程序载入内存并验证程序输出的 VHDL 代码。我们用一个常数数组表示要写入内存的指令，用另一个常数数组表示程序的期望输出，用于和处理处理器执行后的结果进行比较。

然而，注意到处理器和测试平台都连接内存，这就是说测试平台和处理器在同一时间都要控制这两个信号。解决此问题的一种方法是：在内存的输入端口放置多路选择器。我们用三个多路选择器完成此功能：Address_Mux（地址选择），CS_Mux（CS 信号选择），WE_Mux（WE 信号选择）。多路选择器的片选信号为 *init*。当此信号为‘1’时，三个多路选择器选择测试平台的地址、CS 和 WE 信号。当该信号为‘0’时，选择处理器的相关信号和地址。同时，我们在初始化过程中进行 CPU 复位，以确保在测试平台把所有指令写入内存前，CPU 不工作。当 *init* 为‘0’时，CPU 和内存才正常连接工作。

当 MIPS 程序执行时，每个测试指令均在不同的寄存器中存储各自的结果。在所有的测试指令执行完毕后，程序执行一系列存储的指令。这些指令把不同寄存器中的内存送到数据总线上。所以，如果我们要验证 10 条指令，则需要 10 个字的存储指令。在每次存储过程中，总线上的数据均与寄存器期望结果进行比较，我们使用 assert 语句完成此过程。在 MIPS 处理器中，寄存器$0 总为 0，所以我们没有在寄存器文件中对其加以实现。因此，我们使用一个指令把寄存器$0 清零。测试指令序列中的第一个指令就是完成此项工作的。在正常的 MIPS 处理器程序代码中，你不会发现一个指令以存储器$0 为目标寄存器。在 MIPS 中根本忽略对寄存器$0 的写操作。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity MIPS_Testbench is
end MIPS_Testbench;

architecture test of MIPS_Testbench is
  component MIPS
    port(CLK, RST: in std_logic;
         CS, WE: out std_logic;
         ADDR: out unsigned (31 downto 0);
         Mem_Bus: inout unsigned(31 downto 0));
  end component;
  component Memory
    port(CS, WE, CLK: in std_logic;
         ADDR: in unsigned(31 downto 0);
         Mem_Bus: inout unsigned(31 downto 0));
  end component;

  constant N: integer : = 8;
  constant W: integer : = 26;
  type Iarr is array(1 to W) of unsigned(31 downto 0);
  constant Instr_List: Iarr : = (
    x"30000000", -- andi $0, $0, 0   => $0 = 0
    x"20010006", -- addi $1, $0, 6   => $1 = 6
    x"34020012", -- ori $2, $0, 18   => $2 = 18
    x"00221820", -- add $3, $1, $2   => $3 = $1 + $2 = 24
    x"00412022", -- sub $4, $2, $1   => $4 = $2 - $1 = 12
    x"00222824", -- and $5, $1, $2   => $5 = $1 and $2 = 2
    x"00223025", -- or $6, $1, $2    => $6 = $1 or $2 = 22
    x"0022382A", -- slt $7, $1, $2   => $7 = 1 because $1<$2
    x"00024100", -- sll $8, $2, 4    => $8 = 18 * 16 = 288
    x"00014842", -- srl $9, $1, 1    => $9 = 6/2 = 3
    x"10220001", -- beq $1, $2, 1    => should not branch
    x"8C0A0004", -- lw $10, 4($0)    => $10 = 5th instr = x"00412022" = 4268066
    x"14620001", -- bne $1, $2, 1    => must branch to PC+1+1
    x"30210000", -- andi $1, $1, 0   => $1 = 0 (skipped if bne worked correctly)
    x"08000010", -- j 16             => PC = 16
    x"30420000", -- andi $2, $2, 0   => $2 = 0 (skipped if j 16 worked correctly)
    x"00400008", -- jr $2            => PC = $2 = 18 = PC+1+1. $3 wrong if fails
    x"30630000", -- andi $3, $3, 0   => $3 = 0 (skipped if jr $2 worked correctly)
    x"AC030040", -- sw $3, 64($0)    => Mem(64) = $3
    x"AC040041", -- sw $4, 65($0)    => Mem(65) = $4
    x"AC050042", -- sw $5, 66($0)    => Mem(66) = $5
    x"AC060043", -- sw $6, 67($0)    => Mem(67) = $6
    x"AC070044", -- sw $7, 68($0)    => Mem(68) = $7
    x"AC080045", -- sw $8, 69($0)    => Mem(69) = $8
    x"AC090046", -- sw $9, 70($0)    => Mem(70) = $9
    x"AC0A0047"  -- sw $10, 71($0)   => Mem(71) = $10
);
    -- The last instructions perform a series of sw operations that store
    -- registers 3-10 to memory. During the memory write stage, the testbench
    -- will compare the value of these registers (by looking at the bus value)
    -- with the expected output. No explicit check/assertion for branch
    -- instructions, however if a branch does not execute as expected, an error
    -- will be detected because the assertion for the instruction after the
    -- branch instruction will be incorrect.
  type output_arr is array(1 to N) of integer;
  constant expected: output_arr: = (24, 12, 2, 22, 1, 288, 3, 4268066);
  signal CS, WE, CLK: std_logic : = '0';
  signal Mem_Bus, Address, AddressTB, Address_Mux: unsigned(31 downto 0);
  signal RST, init, WE_Mux, CS_Mux, WE_TB, CS_TB: std_logic;
begin
  CPU: MIPS port map (CLK, RST, CS, WE, Address, Mem_Bus);
```

图 9.10 处理器模块的测试程序

```
    MEM: Memory port map (CS_Mux, WE_Mux, CLK, Address_Mux, Mem_Bus);

    CLK <= not CLK after 10 ns;
    Address_Mux <= AddressTB when init = '1' else Address;
    WE_Mux <= WE_TB when init = '1' else WE;
    CS_Mux <= CS_TB when init = '1' else CS;

    process
    begin
      rst <= '1';
      wait until CLK = '1' and CLK'event;

      --Initialize the instructions from the testbench
      init <= '1';
      CS_TB <= '1'; WE_TB <= '1';
      for i in 1 to W loop
        wait until CLK = '1' and CLK'event;
        AddressTB <= to_unsigned(i-1,32);
        Mem_Bus <= Instr_List(i);
      end loop;
      wait until CLK = '1' and CLK'event;
      Mem_Bus <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
      CS_TB <= '0'; WE_TB <= '0';
      init <= '0';
     wait until CLK = '1' and CLK'event;
     rst <= '0';

     for i in 1 to N loop
       wait until WE = '1' and WE'event;  -- When a store word is executed
       wait until CLK = '0' and CLK'event;
       assert(to_integer(Mem_Bus) = expected(i))
         report "Output mismatch:" severity error;
     end loop;

     report "Testing Finished:";
  end process;
end test;
```

图 9.10（续）　处理器模块的测试平台

下列命令语句用于对 VHDL 模块的测试。所有我们感兴趣的信号都不放在最顶层实体中（这里是测试平台）。此时，为了正确的仿真，我们必须提供信号的完整路径（尤其指向元件并在元件中出现的信号）。命令语句 configure list -delta collapse 用于去除中间 Δ 延迟的输出。

```
add list -hex sim:/mips_testbench/cpu/instr
add list -unsigned sim:/mips_testbench/cpu/npc
add list -unsigned sim:/mips_testbench/cpu/pc
add list -unsigned sim:/mips_testbench/cpu/state
add list -unsigned sim:/mips_testbench/cpu/alu_ina
add list -unsigned sim:/mips_testbench/cpu/alu_inb
add list -signed sim:/mips_testbench/cpu/alu_result
add list -signed sim:/mips_testbench/cpu/addr
configure list -delta collapse
run 2330
```

仿真结果如下所示。

MIPS 指令	ns	Instr	PC	State	ALU_InA	ALU_InB	ALU_Result	Addr
andi $0, $0, 0	570	30000000	0	0	–	–	0	0
	908	30000000	1	1	–	–	0	X
	610	30000000	1	2	–	0	0	X
	630	30000000	1	3	–	0	0	0
andi $1, $0, 6	650	30000000	1	0	0	0	0	1

（续表）

MIPS 指令	ns	Instr	PC	State	ALU_InA	ALU_InB	ALU_Result	Addr
	670	20010006	2	1	0	0	0	0
	690	20010006	2	2	0	6	6	0
	710	20010006	2	3	0	6	0	6
ori $2, $0, 18	730	20010006	2	0	0	6	0	2
	750	34020012	3	1	0	18	0	6
	770	34020012	3	2	0	18	18	6
	790	34020012	3	3	0	18	0	18
and $3, $1, $2	810	34020012	3	0	0	18	0	3
	830	00221820	4	1	6	6176	0	18
	850	00221820	4	2	6	18	24	18
	870	00221820	4	3	6	18	0	24
sub $4, $2, $1	890	00221820	4	0	6	18	0	4
	910	00412022	5	1	18	6	0	24
	930	00412022	5	2	18	6	12	24
	950	00412022	5	3	18	6	0	12
and $5, $1, $2	970	00412022	5	0	18	6	0	5
	990	00222824	6	1	6	6	0	12
	1010	00222824	6	2	6	18	2	12
	1030	00222824	6	3	6	18	0	2
or $6, $1, $2	1050	00222824	6	0	6	18	0	6
	1070	00223025	7	1	6	18	0	2
	1090	00223025	7	2	6	18	22	2
	1110	00223025	7	3	6	18	0	22
slt $7, $1, $2	1130	00223025	7	0	6	18	0	7
	1150	0022382A	8	1	6	18	0	22
	1170	0022382A	8	2	6	18	1	22
	1190	0022382A	8	3	6	18	0	1
sll $8, $2, 4	1210	0022382A	8	0	6	18	0	8
	1230	00024100	9	1	0	18	0	1
	1250	00024100	9	2	0	18	288	1
	1270	00024100	9	3	0	18	0	288
srl $9, $1, 1	1290	00024100	9	0	0	18	0	9
	1310	00014842	10	1	0	6	0	288
	1330	00014842	10	2	0	6	3	288
	1350	00014842	10	3	0	6	0	3
beq $1, $2, 1	1370	00014842	10	0	0	6	0	10
	1390	10220001	11	1	6	18	0	3
	1410	10220001	11	2	6	18	−12	3
lw $10, 4($0)	1430	10220001	11	0	6	18	0	11
	1450	8C0A0004	12	1	0	–	0	−12
	1470	8C0A0004	12	2	0	4	4	−12

（续表）

MIPS 指令	ns	Instr	PC	State	ALU_InA	ALU_InB	ALU_Result	Addr
	1490	8C0A0004	12	3	0	4	0	4
	1510	8C0A0004	12	4	0	4	0	4
bne$3,64($0)	1530	8C0A004	12	0	0	4	0	12
	1550	14620001	13	1	24	1	0	4
	1570	14620001	13	2	24	18	6	4
j 16	1590	14620001	14	0	24	18	0	14
	1610	08000010	15	1	0	0	0	6
jr $2	1630	08000010	16	0	0	0	0	16
	1650	00400008	17	1	18	0	0	6
	1670	00400008	17	2	18	0	0	6
sw $3, 64($0)	1690	00400008	18	0	18	0	0	18
	1710	AC030040	19	1	0	24	0	0
	1730	AC030040	19	2	0	64	64	0
	1750	AC030040	19	3	0	64	0	64

在这里，我们没有给出指令载入内存模块的初始周期。从处理器进行指令读取开始给出了数据。这里只给出了第一个存储指令，但是在测试平台中测试了所有存储指令。根据读取的内存中的数据，我们可以做更完整的测试。

本章中，我们介绍了一种流行的 RISC 指令集：MIPS。我们从指令集的规范开始，给出了实现一个 MIPS 指令子集的设计。我们给出了一个可综合的 VHDL 模块。我们说明了测试处理器模块的测试平台的用法。

习题

9.1 ISA 是什么意思？奔腾 4 和奔腾 3 中的 ISA 也是这一意思吗？

9.2 微处理器 *X* 的指令集中有 30 条指令，微处理器 *Y* 的指令集中有 45 条指令。已知 *Y* 为 RISC 处理器，可以得出结论 *X* 也是 RISC 处理器吗？为什么？

9.3 列出 RISC 处理器的 4 个重要特点。

9.4 MIPS 的 addi 指令和 addiu 指令有何区别？

9.5 给出下面的 MIPS 指令的机器语言代码，并且用十六进制表示结果。下面指令中所有的偏置均为十进制数。

(i) add $6, $7, $8
(ii) lw $5, 4($6)
(iii) addiu $3, $2, -2000
(iv) sll $3, $7, 12
(v) beq $6, $5, -16
(vi) j 4000

9.6 给出下面 MIPS 指令的机器语言编码，并且用十六进制表示结果。下面指令中所有的偏置均为十进制数。

(i) addi $5, $4, 4000
(ii) sw $5, 20($3)
(iii) addu $4, $5, $3
(iv) bne $2, $3, 32
(v) jr $5
(vi) jal 8000

9.7 下列十六进制数表示何 MIPS 指令？如果不是表 9.7 中的指令，则认为是非法操作码。

(i) 33333300
(ii) 8D8D8D8D
(iii) 1777FF00
(iv) BDBD00BD
(v) 01010101

9.8 下列十六进制数表示何 MIPS 指令？如果不是表 9.7 中的指令，则认为是非法操作码。

(i) 20202020
(ii) 00E70018
(iii) 13D300C8
(iv) 0192282A
(v) 0F6812A4

9.9 写出下列伪码对应的 MIPS 汇编语言程序。假设 x 和 y 数组的初始位置为 4000 和 8000（十进制）。

```
for(i = 0; i < 100; i++)
  x(i) = x(i) * y(i)
```

9.10 写出下列伪码对应的 MIPS 汇编语言程序。假设 x 和 y 数组的初始位置为 4000 和 8000（十进制）。

```
for(i = 1; i < 100; i++)
  x(i) = x(i) + x(i-1)
```

9.11 写出下列伪码对应的 MIPS 汇编语言程序。假设 x 和 y 数组的初始位置为 4000 和 8000（十进制），*a* 的存储位置为 12000（十进制）。

```
for(i = 0; i < 100; i++)
  y(i) = a * x(i) + y(i)
```

9.12 图 9.8 是一个 MIPS 子指令集模块。用当前 Xilinx 软件进行综合并以 Xilinx FPGA 作为目标器件。需要使用多少个逻辑模块、触发器和存储模块？（注意，也可采用你用的不同公司生产的 FPGA 及其软件来完成此题）

9.13 (a) 图 9.8 是一个 MIPS 子指令集模块。我们可以通过添加模块与 FPGA 开发板的输入开关和 LED 显示器界露以增强该模块。你的接口可以停止 MIPS 处理器的操作，并且可以在个 LED 上显示$1 的低八位。你的接口也可以把开发板的内部时钟进行分频，以供给模块较慢的时钟（如 100 Hz）。你可以根据开发板的功能，使用其他 LED 或显示设备显示其他信息。综合该模块并在开发板上实现它。

(b) 此问中可以使用(a)中的模块。写出一个循环亮灯模块的 MIPS 汇编程序（使用开发板上的 8 个 LED）。从一个 LED 到下一个 LED 依次亮灯，每次亮灯的持续时间为 1 秒。

(c) 此问中可以使用(a)中的模块。写出一个交通灯控制器的 MIPS 汇编程序。按照下面的模式实现该交通灯控制器：

路 A			路 B			
红	黄	绿	红	黄	绿	
0	0	1	1	0	0	5 秒
0	1	0	1	0	0	2 秒
1	0	0	1	0	0	1 秒
1	0	0	0	0	1	5 秒
1	0	0	0	1	0	2 秒
1	0	0	1	0	0	1 秒，然后重复

9.14 很多微处理器通过内存映射实现输入和输出。假设内存单元 F0002F2F 是处理器的一个并行端口。一个方波在并行端口最低位的频率为 8 MHz，写出此方波的 MIPS 程序。现在有一个基于图 9.8 的 MIPS 处理器原型模块，以时钟频率 100 MHz 运行。

9.15 (a) 在 MIPS 的 VHDL 代码子集（参见图 9.8）的 add 和 addi 指令中加入溢出检测。

(b) 写出一个测试平台验证(a)的代码。

9.16 (a) 对于图 9.8 所示 MIPS 子集的所有可溢出指令中加入溢出检测。

(b) 写出一个测试平台验证(a)的代码。

9.17 (a) 在 MIPS 的 VHDL 代码子集中（参见图 9.8）加入 MIPS 指令 `JAL`（跳转和链接）。`JAL` 用于过程调用中。`JAL jumpaddr` 把返回地址(PC + 1)放入寄存器文件$31 中，接着到达 `jumpaddr` 指向的下一条指令（注意，原始 MIPS 使用(PC + 4)和 `jumpaddr*4`；但是在第 9 章中我们使用字地址而不是使用字节地址，所以用'1'代替'4'）。JAL 指令采用 J 格式；因此头 6 位是操作码(3)，剩余的 26 位是 `jumpaddr`。要求对 VHDL 代码做尽量少的改动。

(b) 写出一个测试平台验证该指令。

9.18 (a) 编写一个指令，此指令可以把存储在两个通用寄存器低 16 位的两个数相乘，并把得到的积存储到另外一个 32 位寄存器中，这些寄存器均在图 9.8 所示处理器中（注意，MIPS 中不存在此指令）。

(b) 写出一个测试平台验证该指令。

9.19 （本题可以作为一个项目来使用。流水线的更多内容可以在参考文献[37]中得到）现代微处理器采用流水线以改进指令吞吐量。下面考虑一个 5 级流水线，这 5 级分别为读取级、译码和读寄存器级、执行级、内存访问级和寄存器写回级。在第一级中，我们把指令从指令内存中取出来；在第二级，对读取的指令进行译码，在此级中操作数寄存器也要进行读操作；在第三级，对第二级中寄存器中读取的操作数进行算术或逻辑操作。在第四级，通过载入/存储指令，把数据读出或写入内存。算术指令在本级中不做任何操作。在第五级，算术指令把结果写入目标寄存器。

(a) MIPS 设计的一个流水线实示于现图 9.8。画出流水线基本结构的实现框图。写出 VHDL 代码，使用目标 FPGA 对其进行综合，并在 FPGA 开发板上进行实现。假设每个级都需要一个时钟周期。在开发板上进行实现时，使用 8 Hz 时钟。

假设指令内存访问和数据内存访问均只使用一个时钟周期。因此要把指令和数据内存分开（或者必须有两个端口），这样才可以使第一级和第四级中的访问同时进行。

只要一个指令与未完成指令（流水线中前一个指令）相互独立，则它就可以在第二级从寄存器文件中读取操作数。如果二者存在相关性，则在译码级当前指令必须等待，直到寄存器数据准备好为止。必须对每条指令及其前一条指令测试相关性。此操作可以通过比较当前指令的源寄存器和未完成指令的目标寄存器加以实现。

在第五级对寄存器文件进行写操作，在第二级对其进行读操作。由此，我们可以得到这样一个合理的假设：在头半个周期内进行写操作，在后半个周期内进行读操作。假设在头半个周期中写入的数据可以在后半个周期内读取。

(b) 要执行 N 个相互独立的指令需要使用多少个时钟周期？

(c) 在流水线中实现下面的指令需要使用多少个时钟周期？

```
add $5,$4,$3
add $6,$5,$4
add $7,$6,$5
add $8,$7,$6
```

9.20 （本题可以作为一个项目来使用。关于流水线和数据前置（data forwarding）的更多的内容参考文献 37 中可以得到。）在题 9.19 中，我们假设数据应在指令写回级写入寄存器文件中，然后后续指令对其进行读操作。这样，若指令 *i*+1 与指令 *i* 相关，则引入两个操作周期。很多处理器均采用数据前置技术解决这一问题。如果一条指令需要使用其前一条指令的结果，则结果会前置到当前指令。该操作可以通过在 ALU 输入端加入 MUX 实现，此 MUX 决定从何处得到操作数：是从寄存器文件中，还是从 ALU 输出前送路径中，或者从内存访问级（第四级）输出中。若可以明确指令之间的关联性，则就通过 MUX 确保前送数据的正确性。

(a) 基于数据前置实现流水线结构的图 9.8 的 MIPS 设计。画出前置硬件框图，写出 VHDL 代码，综合该代码，并用 FPGA 开发板加以实现。在开发板上实现时，使用 8 Hz 时钟。

(b) 比较题 9.10、题 9.11 代码的执行周期数，比较习题 9.19 和图 9.8 代码的执行周期数。



第 10 章　硬件测试和可测试性设计

本章介绍数字系统测试及其使系统容易测试的设计方法。我们已经介绍了设计过程中的测试。我们写出了 VHDL 测试平台，用于验证总系统和算法的正确性。我们在逻辑层面对设计进行仿真并验证其是否逻辑正确和满足规范。IC 逻辑层设计结束后，我们将在电路层面对设计进行检测，验证设计是否得到正确实现，时序是否正确。

当一个数字系统已经制造完毕后，仍需要对其进行测试以验证其功能的正确性。我们把制造完毕的 IC 复制多份，且每个复制后的 IC 都要通过测试验证其性能与制造工艺缺陷无关。此测试是很费时和昂贵的。因此，在现代复杂 IC 制造中，测试费用占整个制造费用的大部分。所以，寻找测试数字系统的更有效方法是非常重要的。在现代 IC 设计中，可测试设计(**DFT**)是一个很重要的方面。

本章中，我们首先针对经常出现的故障，讨论组合逻辑电路的测试方法。随后，介绍如何确定时序逻辑的测试序列。我们还将介绍自动测试模式生成器(**ATPG**)，它可以生成待测电路和系统的测试序列。在测试时，我们会碰到这样一个问题：我们只能用到待测电路的输入和输出，却不能触到内部状态。为了解决这个问题，我们可以把内部测试点引到 IC 芯片的管脚上。为了减少所需测试管脚的数量，我们引入了一个概念：扫描设计，它可以把系统的所有状态存到移位寄存器中，并且串行移出。最后，我们讨论内嵌自测试(**BIST**)概念。只要在 IC 上加入更多的元件，我们就可以内部生成测试序列，并验证电路对这些序列的响应，而不需要昂贵的外部测试。

10.1　组合逻辑电路的测试

两个常见的故障是短路和开路。如果一个门的输入接地，则相当于此输入为逻辑 0；如果一个门的输入与正电压相连，则相当于此输入为逻辑 1；如果一个门的输入是开路的，则此输入既可以为 0 也可以为 1，这是由逻辑电路的类型决定的。这样，逻辑电路的故障经常模拟为陷 0 故障（stuck at 0, 陷 0）和陷 1 故障（stuck at 1, 陷 1）。为了检测一个门输入的陷 0 故障，门输入必须为 1，这样可以检测出任何 0 的改变。同样，对为了检测一个门输入的陷 1 故障，门输入必须为 0，这样可以检测出任何 1 的改变。

我们通过设置所有输入为 1，可以检测一个与门是否存在陷 0 故障，如图 10.1(a)所示。输出的正确值应为 1，但是如果任何输入存在陷 0 态，则输出为 0。图中输入 a 处的标记$1 \to 0$表示 a 的正常输入为 1，但是由于陷所以它变为 0。输出处$1 \to 0$表示这一变化已经传输到输出。在测试一个与门是否存在陷 1 时，我们可以把所要测试的输入赋 0，其他输入赋 1，如图 10.1(b)所示。输出的正确值为 0，但是如果被测输入存在陷 1，则输出为 1。在测试 OR 门的输入是否存在陷 1 时，我们可以把所有输入都赋 0，如果任何输入存在陷 1 态，则输出为 1，如图 10.1(c)所示。在测试一个 OR 门是否存在陷 0 故障时，我们可以把所要测试的输入赋 1，其他输入赋 0。输出的正确值为 0，但是如果被测输入存在陷 0 态，则输出为 0，如图 10.1(d)所示。在测试门输入是否存在陷 0 和陷 1 的过程中，我们也可以对输出的陷 0 和陷 1 进行测试。

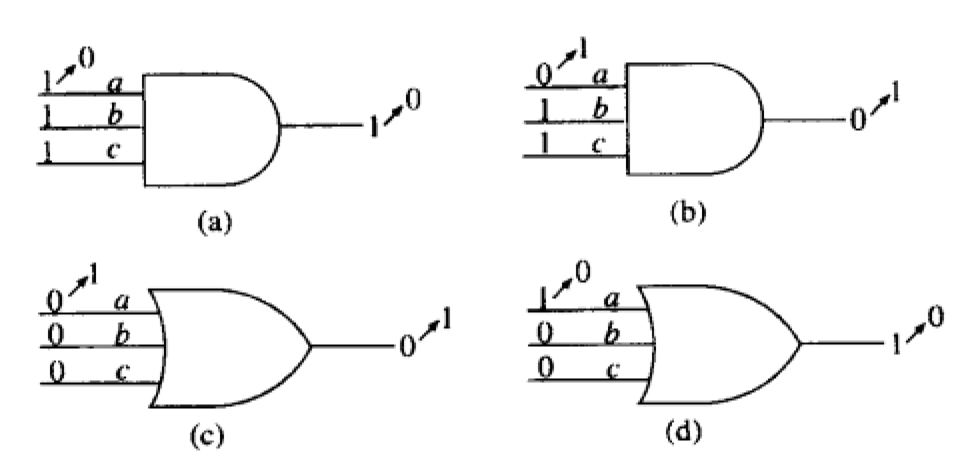


图 10.1　测试 AND 门和 OR 门的陷入故障

图 10.2 所示 AND-OR 电路具有 9 个输入和 1 个输出。我们假设 OR 门输入(p, q, r)不可访问，所以我们不能对两个门进行单独测试。测试此电路的一个方法就是把 $2^9 = 512$ 个不同输入组合都进行测试，并观察其输出。另一种更有效的方法就是对所有的陷 0 和陷 1 进行检测。如表 10.1 所示，对 AND 门输入 a, b, c 测试其是否存在陷 0 时，我们把 a, b, c 都赋 1，如图 10.2(a)所示。如果任何门输入有陷 0，则门输出(p)将变为 0。为了使这一变化传到 OR 门输出，则其他 OR 门输入必须为 0。为了实现这点，我们可以令 d =0, g =0（e, f, h 和 i 为任意值），测试矢量对 p0（p 陷 0）. a0. b0 和 c0 进行检测。再令 d =e =f =1 且 a=g =0，对 d0, e0, f0 和 q0 进行检测，然后进行第三次测试，令 $g = h = i = 1$ 且 $a = d = 0$，对其他陷 0 进行测试。对输入 a 测试其是否存在陷 1 时(a1)，我们令 a =0, b =c =1，如图 10.2(b)所示。如果 a 存在陷 1，则门输出(p)将变为 1。为了使此改变传到输出，则必须有 q =r=0。但是如果令 d=g=0，且 e=f=h =i=1，则可以在测试 a1 的同时，对 d1 和 g1 进行测试。而且可以用同样的测试矢量对 p1，q1 和 r1 进行测试。从表中可以看出，在测试 b1，e1 和 h1 时，我们使用单一测试矢量，在测试 c1, f1 和 i1 时也一样。这样，我们可以只用 6 次测试就能够对所有的陷 0 和陷 1 进行检测，但是遍历测试法则需要 512 次测试。当进行完 6 次测试时，我们就可以确定是否有故障存在，但是我们不能确定故障出现的精确地址。在之前的分析中，我们假设一次只有一个故障出现，然而，在很多情况下，我们也需要对多个故障进行检测。

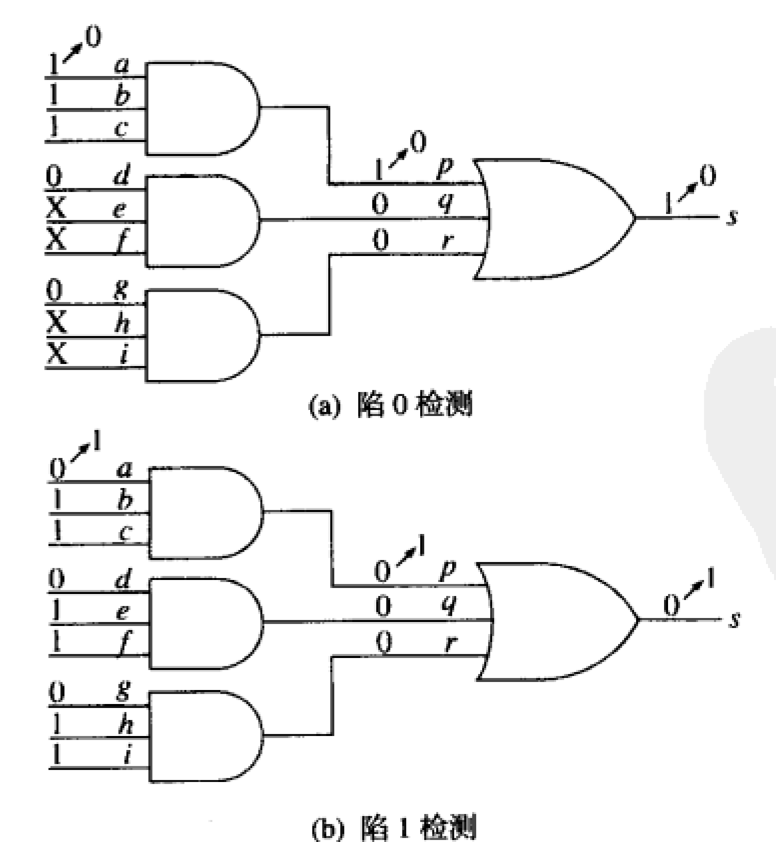


图 10.2　对 AND-OR 电路进行测试

表 10.1　图 10.2 的测试矢量

a	b	c	d	e	f	g	h	i	检测对象
1	1	1	0	X	X	0	X	X	a0, b0, c0, p0
0	X	X	1	1	1	0	X	X	d0, e0, f0, q0
0	X	X	0	X	X	1	1	1	g0, h0, i0, r0
0	1	1	0	1	1	0	1	1	a1, d1, g1, p1, q1, r1
1	0	1	1	0	1	1	0	1	b1, e1, h1, p1, q1, r1
1	1	0	1	1	0	1	1	0	c1, f1, i1, p1, q1, r1

对多级电路的测试要比测试两级电路的测试复杂得多。为了测试电路内部的故障，我们必须选择一组输入去激励故障，并把故障响应传输到输出端。图 10.3 中电路的输入为 *a, b, c, d* 和 *e*。如果我们要对门输入 *n* 测试其是否存在陷 1，则 *n* 必须赋值为 0。为此令 *c*=0, *a*=0 和 *b*=1，如图 10.3 所示。为了把 *n* 的陷 1 故障传递到输出 *F*，我们必须令 *d*=1, *e*=0。对于这一组输入，如果 *a, m, n* 或 *p* 存在陷 1 故障，则输出值将不正确，此时故障就能检出。另外，如果我们把 *a* 的赋值由 0 变为 1，且门输入 *a, m, n* 或 *p* 中存在陷 0 故障，则输出 *F* 将由 1 变为 0。此时我们说通过 *a, m, n* 和 *p* 的路径敏化了，因为该路径上的任何故障都可以被检测出。路径敏化(path sensitization)使我们可以只使用一组电路输入就可以对大量不同的陷入故障进行检测。

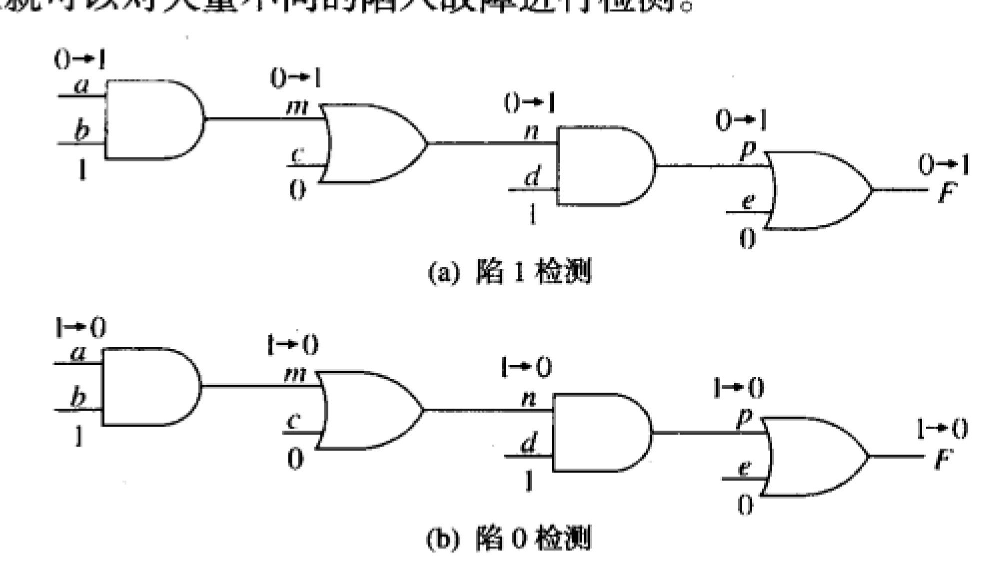


图 10.3　使用路径敏化进行故障检测

下面，我们试着确定用来测试图 10.4 电路中所有的陷 1 和陷 0 的最小测试矢量。假设我们有输入 *A, B, C* 和 *D*，观察输出为 *F*，且内部门电路输入和输出不可访问。确定测试矢量的基本步骤如下：

1. 选择一个未被测试的故障。
2. 确定所需的输入 *ABCD*。
3. 确定其他测试到的故障。
4. 重复此步骤直到找到能测试所有的故障的测试矢量。

让我们从测试输入 *p* 的陷 1 开始。为此，我们必须选择输入 *A, B, C* 和 *D*，使 *p*=0，并且把 *p* 的陷 1 故障传递到输出 *F* 以便观察到。为了传递此故障，我们必须令 *c*=0，*w*=1。为了使 *w*=1，我们令 *t*=1 或 *u*=1。为了使 *u*=1，我们令 *D*=1 且 *r*=1。幸运的是，如果我们令 *C*=0，则 *r*=1。为了使 *p* =0，我们令 *A*=0。再令 *B*=1，我们就可以对路径 *A-a-p-v-f-F* 进行敏化。因此，当设定输入 *ABCD* = 0101 时，我们就可以对 *a*1, *p*1, *v*1 和 *f*1 进行测试。此输入序列也可以对 *c* 陷 1 进行检测。假设 c 陷 1 是门电路的内部故障，所以如果 *c* 存在陷 1 故障，则 *q*=0 且 *r*=1。

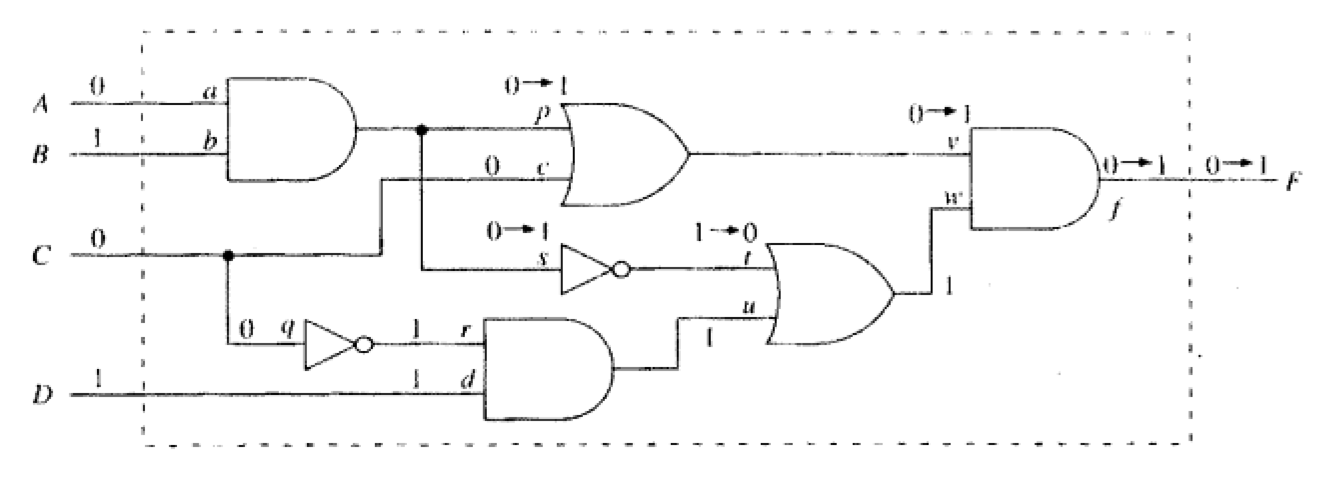


图 10.4 陷入故障测试实例（*p* 的陷 1 测试）

在测试路径 *A-a-p-v-f-F* 是否存在陷 0 故障时，我们可以令 *ABCD* =1101。此输入序列不仅可以对 *a*0, *p*0, *v*0 和 *f*0 进行检测，而且能够对 *b*0, *w*0, *u*0, *r*0, *q*1 和 *d*0 进行检测。为了确定其他陷入故障，我们选择一个未被检测的故障，并确定所需输入 *ABCD* 的值，然后确定需要检测的其他故障。随后反复重复此过程直到所有的故障均被检测。表 10.2 中列出了 5 个测试矢量用于测试图 10.4 所示电路是否存在陷入故障。

表 10.2 图 10.4 所示电路固定故障的测试

				门输入													检测对象
A	*B*	*C*	*D*	*a*	*b*	*p*	*c*	*q*	*r*	*d*	*s*	*t*	*u*	*v*	*w*	*F*	
0	1	0	1	0	1	0	0	0	1	1	0	1	1	0	1	0	*a*1 *p*1 *c*1 *v*1 *f*1
1	1	0	1	1	1	1	0	0	1	1	1	0	1	1	1	1	*a*0 *b*0 *p*0 *q*1 *r*0 *d*0 *u*0 *v*0 *w*0 *f*0
1	0	1	1	1	0	0	1	1	0	1	0	1	0	1	1	1	*b*1 *c*0 *s*1 *t*0 *v*0 *w*0 *f*0
1	1	0	0	1	1	1	0	0	1	0	1	0	0	1	0	0	*a*0 *b*0 *d*1 *s*0 *t*1 *u*1 *w*1 *f*1
1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	0	0	*a*0 *b*0 *q*0 *r*1 *s*0 *t*1 *u*1 *w*1 *f*1

除了陷入故障，还有其他种类的故障，如桥接（bridging）故障。当两个不连接的信号线距离过近时，就会出现桥接故障。对于大型组合逻辑电路，想找到测试所有可能故障的最小测试矢量是非常困难和耗时的。对于具有冗余门的电路，我们不能对某些故障进行测试。即使我们找到一个完备的测试矢量集，但是把所有的测试矢量都进行测试是很耗时和昂贵的。由于这些原因，通常我们选用一个较小的测试矢量集对大部分的故障进行检测。为了生成该测试矢量集，人们开发了很多算法和相应的计算机程序。计算机程序用来对有故障的电路进行模拟，并且对于给定的输入矢量集，允许用户决定对百分之几的故障进行检测，此百分比被称为测试矢量的覆盖率。

10.2 时序逻辑电路的测试

时序逻辑的测试要比组合逻辑的测试麻烦得多，这是因为我们必须使用输入序列用于测试。如果我们只观察输入和输出序列，而不对时序电路中触发器的状态加以观察，则也需要大量的测试序列。基本上，问题在于确定待测时序电路是否与功能正确的电路等效。我们假设待测时序电路具有复位输入，所以我们可以把它复位为已知的初始状态。如果我们要用穷举的法对电路进行测试，则需要复位电路到初始状态，再一个输入测试序列并观察输出序列。如果输出序列正确，则接下来我们需要对其他所有的输入序列进行测试。我们需要对所有状态和状态转换进行检测，此工作量是很大的。由于穷举法完全不可能实现，所以出现了这样一个问题：我们能否找到一个相对小的测试矢量集对电路进行适当的测试？

得到该测试序列的一种方法是把时序电路转化为迭代电路。迭代电路是指时序电路的组合部

分重复多次，以指出每个时刻电路组合部分的条件。由于迭代电路是组合逻辑电路，所以我们可以使用组合电路的标准方法得到迭代电路的测试矢量。

图 10.5 给出了一个标准的 Mealy 时序电路及其相应的迭代电路。图中 X, Z 和 Q 可以为单值也可以为矢量。对于用于时序电路中的组合电路，迭代电路具有 $k+1$ 个完全相同的复制电路，其中 $k+1$ 表示用于测试时序电路的序列长度。在时序电路中，$X(t)$ 代表时间输入序列。在迭代电路中，$X(0)\,X(1)\cdots X(k)$ 表示相同序列的空间输入。迭代电路的每个单元均根据 $Q(t)$ 和 $X(t)$ 计算 $Z(t)$ 和 $Q(t+1)$。最左边的单元计算 $t=0$ 时刻的值，下一个单元计算 $t=1$ 时刻的值，依次类推。在得到迭代电路的测试矢量后，这些矢量作为输入序列用于测试原始时序电路。迭代电路中单元的个数与待测试时序电路所需测试序列的长度有关。

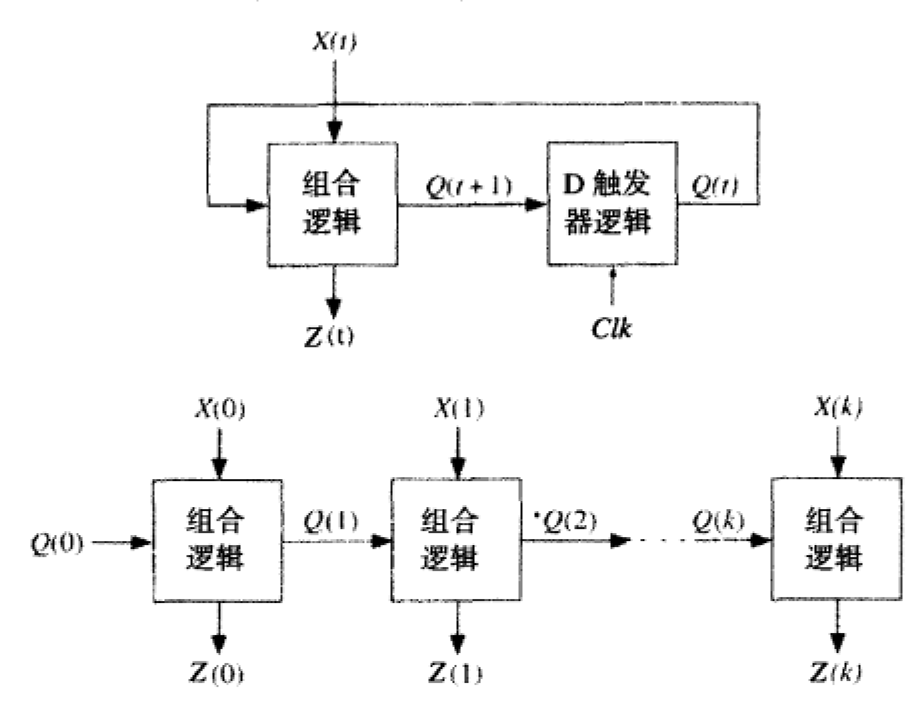


图 10.5 时序和迭代电路

用于测试时序电路的较小的测试序列集是很难得到的。下面考虑图 10.6 所示状态图及其相应的状态表（参见表 10.3）。假设在状态 S_0 电路复位。同时测试序列可以使电路历经所有可能的状态转移，但是这仍不是一个完备的测试。例如，输入序列

$$X = 010110011$$

把图中与此状态相连所有的弧线遍历，并得到输出序列

$$Z = 001011110$$

如果我们把状态 S_3 到 S_0 的弧线变为 S_3 的自循环，如图 10.6 中虚线所示，则输出序列相同，但是改变后的时序机与原来的时序机不等效。

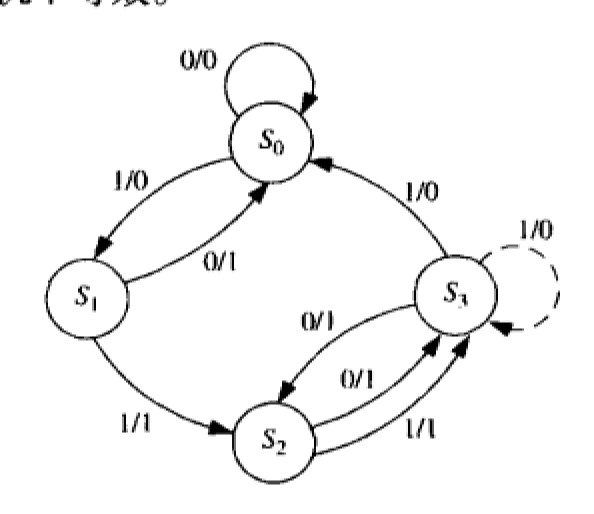


图 10.6 测试实例状态图

如果一个状态图中的每个状态均可以从其他状态转移得到，则此连接称为强连接（strongly connected）。具有强连接的时序电路常用的测试策略为：首先找到能够把每个状态与其他状态区分开的输入序列。这种序列称为区分序列。一个有限输入序列应用于状态机 M 时。当且仅当对于此输入序列可以得到不同的输出时，M 的两个状态才可以分辨。如果对于每个可能的输入序列输出序列是相同的，则显然这两个状态等效。通过证明可以得到：如果 M 的两个状态可以区分，则它们可以被长度为 $n-1$ 或更短的序列区分开来，其中 n 为 M 中状态的个数[28]。对于给定的区分序列，状态表中的每个入口都可以进行验证。

图 10.6 所示实例的一个区分序列为 11。此区分序列可以通过以下步骤得到。首先把状态 S_0, S_1, S_2 和 S_3 分为两组。如果测试序列只有 1 位长，则每组的状态均等效。例如，表 10.3 中应用了长度为 1 位的测试序列，我们可以区分出组$\{S_0, S_3\}$和$\{S_1, S_2\}$。若输入为 1，则$\{S_0, S_3\}$的输出为 0，$\{S_1, S_2\}$的输出为 1。如果测试序列仅仅为 1 时，每组中的状态都是等效的。现在，从表 10.3 中我们可以看出：如果我们再次使用测试序列 1，则组$\{S_0, S_3\}$和$\{S_1, S_2\}$中的所有状态可以被区分出来。因此序列 11 可以区分这四个状态。在最坏的情况下，我们要使用三位序列才能够区分出状态机中的四个状态。当输入序列为 11 时，如果我们从状态 S_0 开始，则输出为 01；如果从 S_1 开始，则输出为 11；如果从 S_2 开始，则输出为 10；如果从 S_3 开始，则输出为 00。这样，通过使用输入序列 11，我们可以把四个状态区分开来。下面我们将使用下列序列对状态表的每个入口进行验证，其中 R 表示回到 S_0 状态。

输入	输出	转移验证
R 0 1 1	0 0 1	$(S_0 \sim S_0)$
R 1 1 1	0 1 1	$(S_0 \sim S_1)$
R 1 0 1 1	0 1 0 1	$(S_1 \sim S_0)$
R 1 1 1 1	0 1 1 0	$(S_1 \sim S_2)$
R 1 1 0 1 1	0 1 1 0 0	$(S_2 \sim S_3)$
R 1 1 1 1 1	0 1 1 0 0	$(S_2 \sim S_3)$
R 1 1 0 0 1 1	0 1 1 1 1 0	$(S_3 \sim S_2)$
R 1 1 0 1 1 1	0 1 1 0 0 1	$(S_3 \sim S_0)$

表 10.3 图 10.6 的状态表

Q_1Q_2	状态	下一状态 $X=0$	1	输出 $X=0$	1
00	S_0	S_0	S_1	0	0
10	S_1	S_0	S_2	1	1
01	S_2	S_3	S_3	1	1
11	S_3	S_2	S_0	1	0

另外一种获得测试序列的方法是基于固定故障的测试。图 10.6 的实现见图 10.7，状态赋值如下：$S_0=00, S_1=10, S_2=01$ 和 $S_3=11$。如果要测试 a 是否存在陷 1 故障，则我们必须首先在状态 S_1 对故障进行激活，即 $Q_1Q_2=10$,, 且设定 $X=0$。在操作正常时，其下一状态为 S_0。但是，如果 a 存在陷 1，则下一状态为 $Q_1Q_2=01$，即为 S_2。测试序列构造如下：

- 到达状态 S_1：$X=1$ 并随后复位。
- 检测 a 是否存在陷 1 故障：$X=0$。
- 区分到达的状态：$X=11$。

最终序列为 R1011。正常的输出序列为 0101，错误输出序列为 0110。

我们通过简单的例子介绍了获得时序电路测试序列的一些方法。当电路中的输入和状态数量

增加时，所需测试序列的数量和长度将快速增加，这样测试序列的获得就更加困难。换句话说，电路测试所需的时间和花费随着待测电路输入和状态的增加而快速增加。

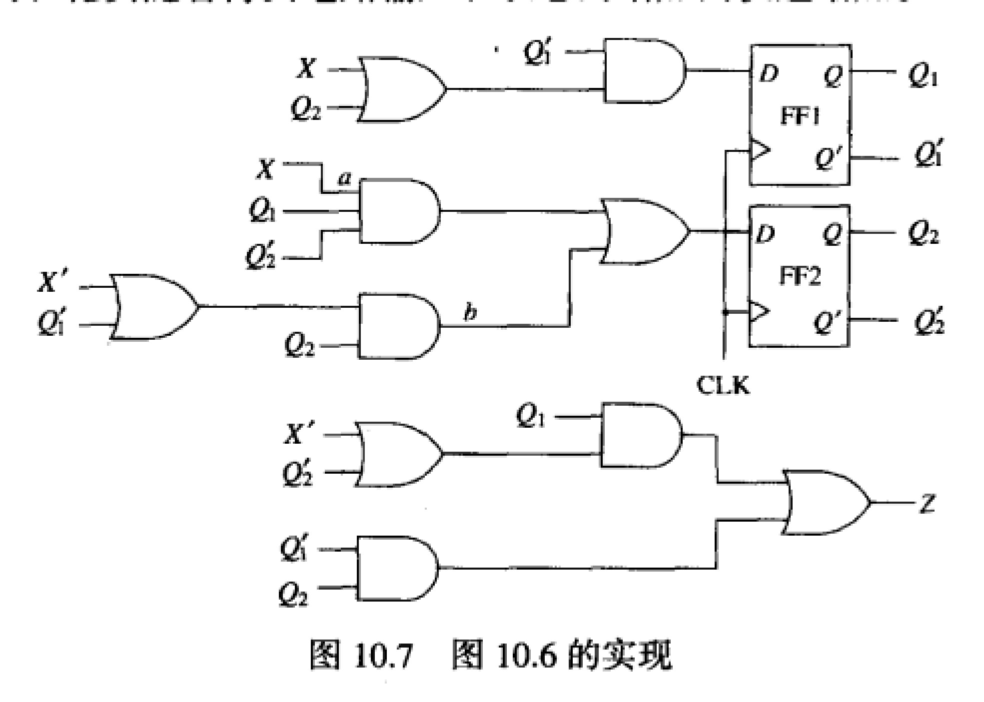


图 10.7　图 10.6 的实现

10.3　扫描测试

如果我们可以观察所有触发器的状态而不是观察电路的输出，则时序电路的测试问题就可以得到很大的简化。对于触发器的每个状态和每个输入组合，我们需要验证电路的输出是否正确，验证电路是否可以到达正确的下一状态。有一种方法是把待测 IC 中每个触发器的输出连接到一个管脚上。由于 IC 的管脚数是非常有限的，所以该方法不是很实用。在不使用很多 IC 管脚的基础上，我们怎样才能观察所有触发器的状态？若这些触发器能排成一个移位寄存器，则我们可以只使用一个 IC 串行输出管脚逐位移出状态信息。这就引出了扫描路径测试的概念。

图 10.8 给出了一种基于两端口触发器的扫描路径测试方法。通常在测试时，我们会把时序电路分成一个组合逻辑部分和一个由触发器构成的状态寄存器。每个触发器均具有两个 D 输入和两个时钟输入。当 $C1$ 接收到脉冲时，输入 $D1$ 存于触发器中。当 $C2$ 接收到脉冲时，输入 $D2$ 存于触发器中。每个触发器的输出 Q 均与下一个触发器的输入 $D2$ 相连，由此构成一个移位寄存器。下一状态($Q_1^+Q_2^+\ldots Q_k^+$)由组合逻辑电路生成，而且在 $C1$ 接收到脉冲时被载入到触发器中。新状态($Q_1Q_2\ldots Q_k$)反馈回组合逻辑电路。当电路未被测试时，使用系统时钟($SCK = C1$)。输入集合为($X_1X_2\ldots X_k$)时，生成输出($Z_1Z_2\ldots Z_m$)。在 SCK 接收到脉冲作用下，电路转到下一状态。

当电路进行测试时，利用扫描数据信号(SDI)和测试时钟(TCK)，可以把状态码移入到寄存器中，以设置为特定的状态。通过输入测试矢量 ($X_1X_2\ldots X_k$)后，验证其输出(($Z_1Z_2\ldots Z_m$)，在 SCK 脉冲作用下，电路到达下一状态。随后，给 TCK 提供脉冲信号，并通过扫描数据输出（SDO）把状态码移出扫描数据寄存器，以验证下一状态是否正确。此方法把时序电路的测试化简为组合电路的测试。因此我们可以使用组合电路测试的任何标准方法生成测试矢量集。由于组合逻辑电路有 n 个 X 输入和 k 个状态输入，所以每个测试矢量具有(($n+k$))位。测试矢量的 X 部分可以直接应用，Q 部分要通过 SDI 进行移位后再使用。总之，测试步骤如下：

1. 使用测试时钟 TCK，并通过 SDI，在测试矢量中扫描 Q_i 的值。
2. 输入相应的 X_i 的测试值。
3. 等待足够的时间，使信号传播到组合电路并验证输出 Z_i 的值。

4．在系统时钟 SCK 上应用一个时钟脉冲，并把 Q_i^+的新值存入到相应的触发器中。

5．在测试时钟 TCK 上应用时钟脉冲，找到并验证 Q_i 的值是否正确。

6．对每个测试矢量都要重复步骤 1～5。

由于在扫描输出前一测试结果的同时可以扫描输入另一个测试矢量，所以步骤 5 和步骤 1 可以重合并行。

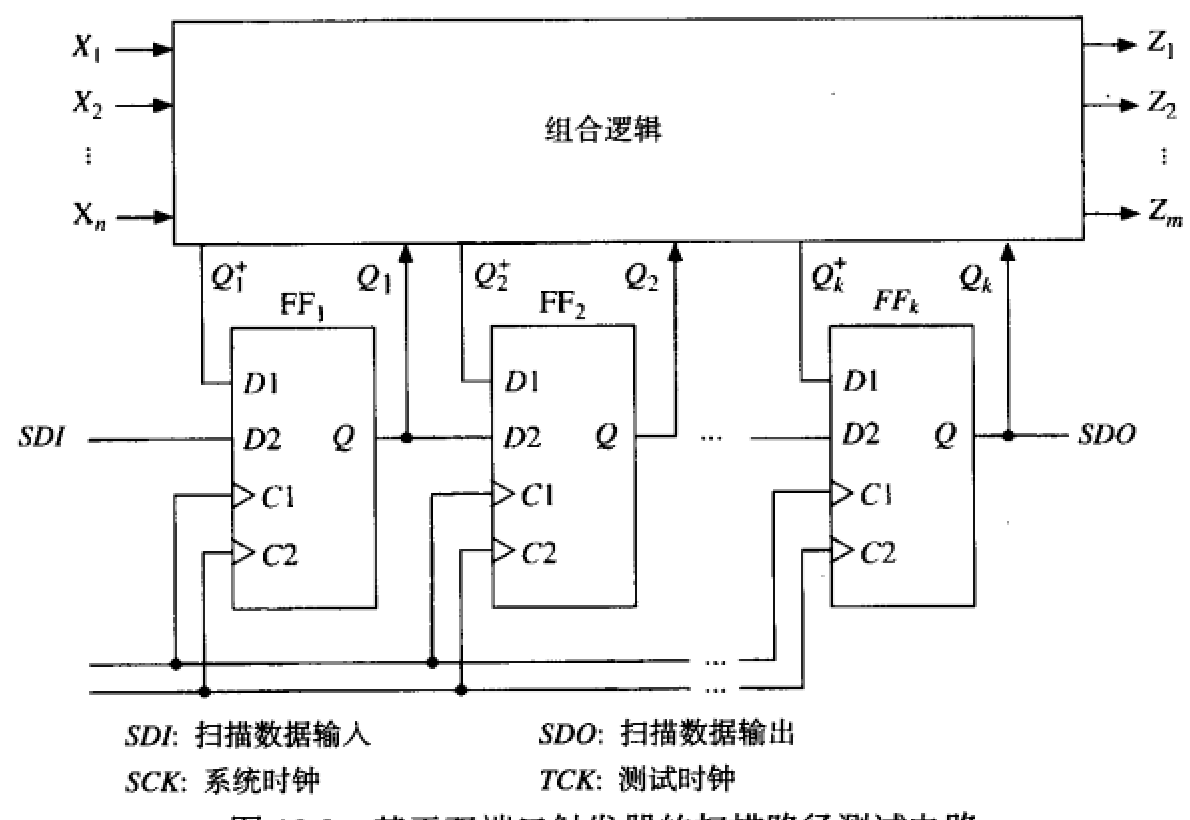
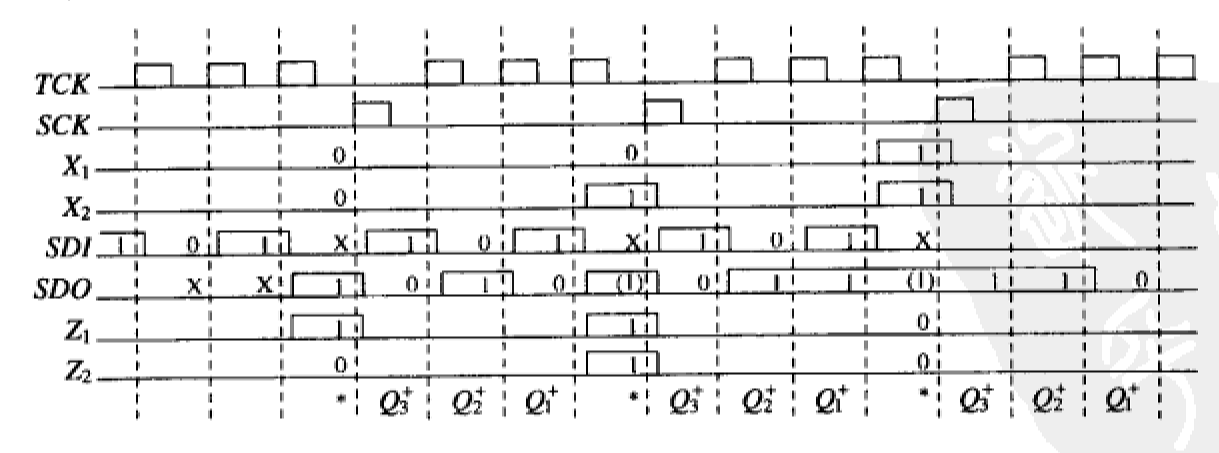


图 10.8　基于双端口触发器的扫描路径测试电路

下面我们把此方法用于一个时序电路中，此电路具有 2 个输入 X_1X_2，3 个触发器 $Q_1Q_2Q_3$ 和 2 个输出 Z_1Z_2，如图 10.8 所示。此电路的一行状态转移表如下所示：

$Q_1Q_2Q_3$	$Q_1^+Q_2^+Q_3^+$ X_1X_2 = 00	01	11	10	Z_1Z_2 00	01	11	10
101	010	110	011	111	10	11	00	01

图 10.9 给出了这一行状态转移表的测试时序。通过 TCK，先把 101 移入，从最低有效位开始（Q_3）。当输入 X_1X_2=00 时，读出输出 Z_1Z_2=10。给 SCK 一个输出脉冲，电路变为状态 010。通过 TCK 把 010 移出，再移入 101 用于下一个测试。这一过程一直延续到测试完毕。



*读取输出（没有给出其他时刻的输出）

图 10.9　扫描测试时序图

一般来说，使用 IC 实现的数字系统都是由组合逻辑模块和触发器构成的，如图 10.10(a)所示。为了在 IC 上应用扫描测试，我们需要把图 10.10(a)中的触发器用双端口触发器代替（或者为其他

类型的可扫描触发器），并把所有的触发器连接成一个扫描链路，如图 10.10(b)所示。然后，我们就可以扫描测试数据到所有的寄存器中，应用测试时钟，最终得到扫描结果。

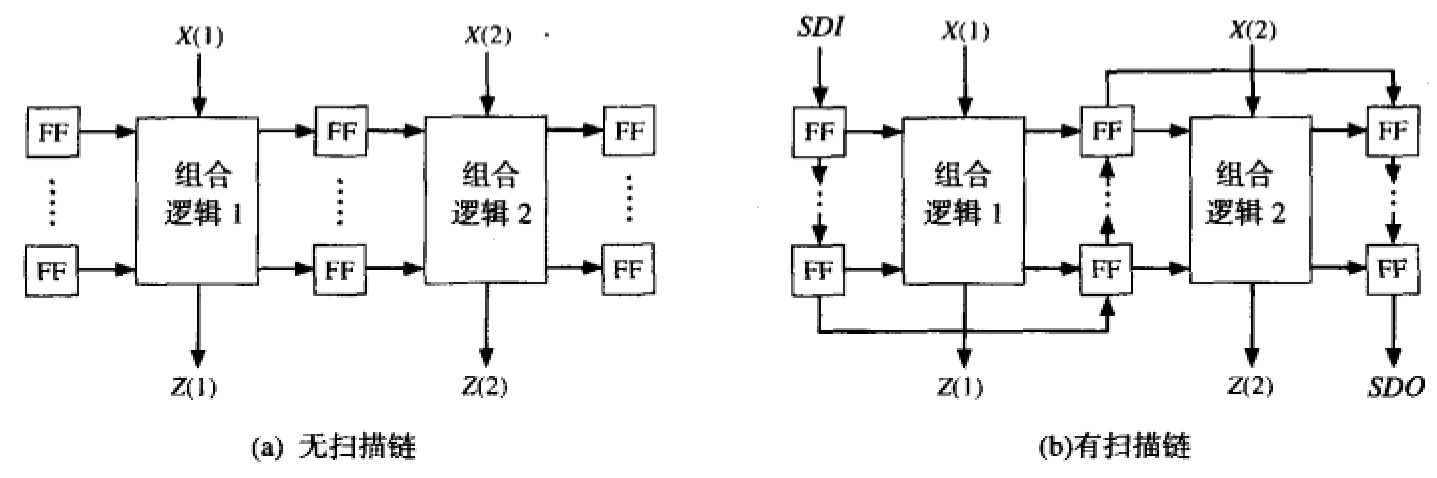


图 10.10 由触发寄存器和组合逻辑模块构成的系统

在一块 PC 板上有多个 IC，那么就把每个 IC 的扫描寄存器连接起来，这样整个板就可以用一个串行存取端口进行测试（参见图 10.11）。

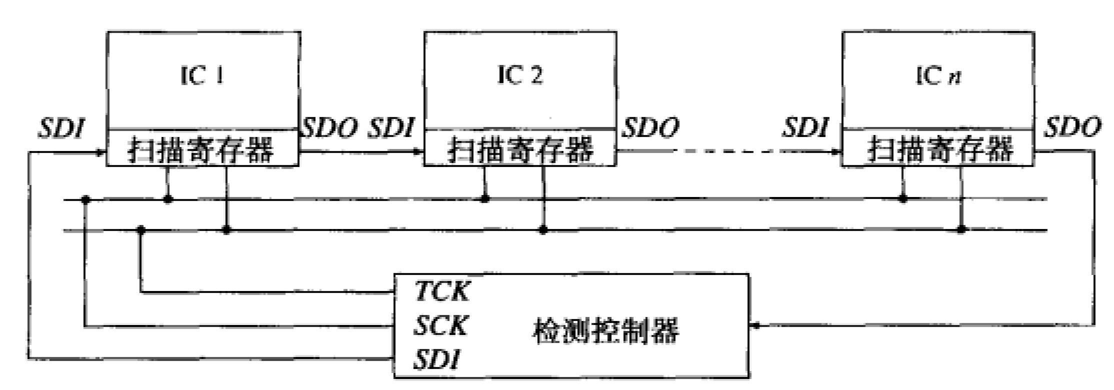


图 10.11　多个 IC 的扫描测试配置

10.4　边界扫描

由于现在 IC 具有越来越多的管脚，也变得越来越复杂，所以印制电路板要有多个层次和更细的走线。对这些含有复杂 IC 的 PC 板进行测试就变得非常困难。通过边界连接信息来对一块板进行测试需要大量的工作并且需要很长的测试序列。当 PC 板的密度较低并且具有较宽的走线时，常常借助于针盘式夹具(bed-of-nails test fixture)。该方法用尖探头接触电路板的走线，这样就可以得到板上各个 IC 的测试数据了。对于具有细走线和复杂 IC 的高密度 PC 板来说，针盘式测试方法是不适用的。

在对复杂 PC 板进行测试时，我们使用边界扫描测试法。这是一种测试具有很多 IC 的电路板的综合方法。联合测试工作组(JTAG)提出了一个边界扫描测试标准："标准测试访问端口和边界扫描结构"。此标准被 ANSI/IEEE 标准 1149.1 采用。很多 IC 生产厂商制造的 IC 芯片都遵循该标准。这样的 IC 芯片可以在一个 PC 板上连接在一起，因此仅使用 PC 板边界连接器中的几个管脚就可以对它们进行测试。

图 10.12 中的 IC 具有 IEEE 标准的边界扫描逻辑。边界扫描寄存器(BSR)中有一个单元放置在每个输入或输出管脚与内部核心逻辑之间。4～5 个 IC 管脚用做测试访问端口(TAP)。TAP 控制器和其他的测试逻辑同样也加载在 IC 的内部核心逻辑上。TAP 管脚的作用（根据标准）如下所示：

TDI 测试数据输入（此数据被串行地移入 BSR 中）
TCK 测试时钟
TMS 测试模式选择
TDO 测试数据输出（从 BSR 中串行输出）
TRST 测试复位（TAP 控制器的复位和测试逻辑；可选管脚）

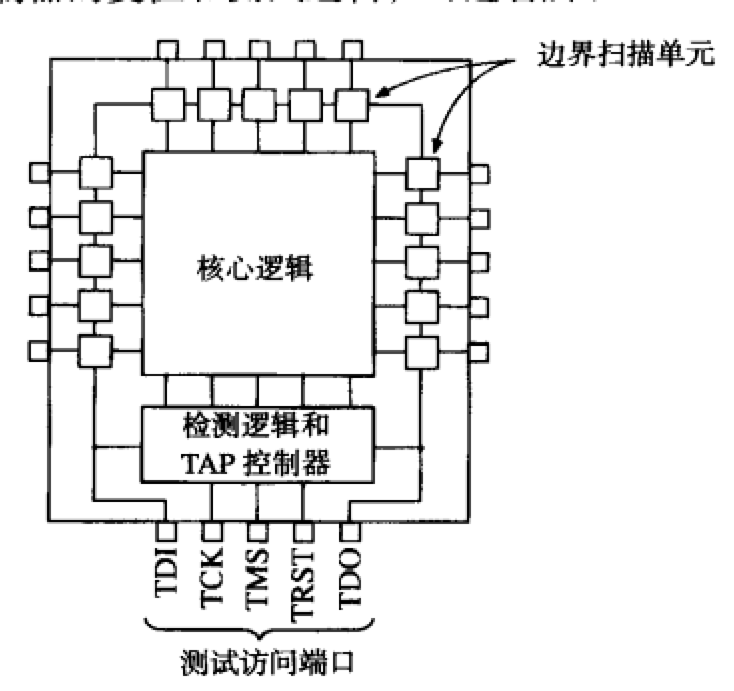


图 10.12 具有边界扫描寄存器和测试访问端口的 IC 芯片

一块具有多个边界测试 IC 的 PC 板见图 10.13。IC 中的边界扫描寄存器是串行连接的，其输入为 *TDI*，输出为 *TDO*。对于所有的 IC，*TCK*, *TMS* 和 *TRST* 是并行连接的。使用这些信号，测试指令和测试数据就可以通过时钟进入到板上的每个 IC 芯片中。

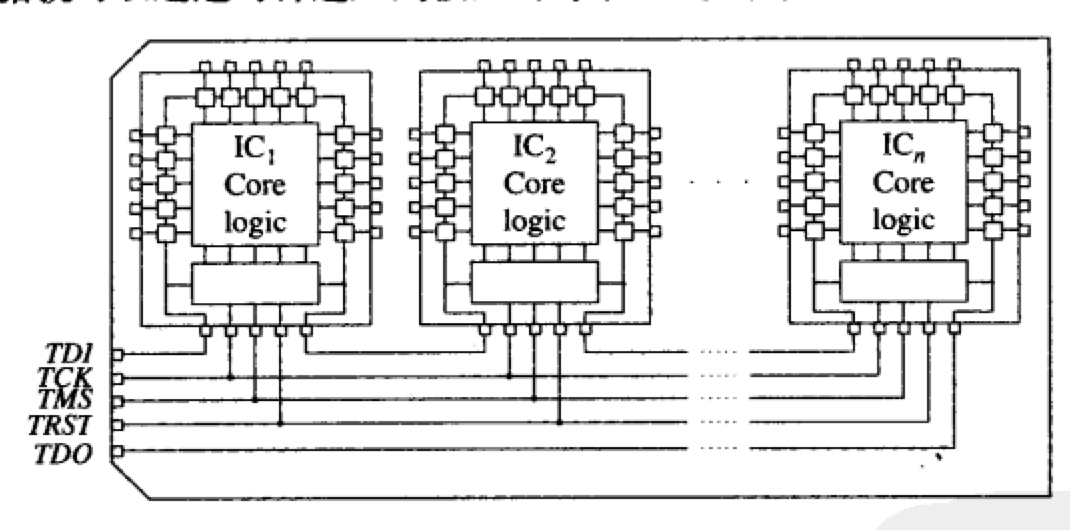


图 10.13 具有边界扫描 IC 的 PC 板

图 10.12 展示了每个 IC 中边界扫描单元的排列，此排列是基于边界扫描标准的。典型的边界扫描单元结构见图 10.14。每个边界扫描单有两个输入：TDI 串行输入和并行输入管脚。而且它还有两个输出：串行输出和并行输出。在正常工作模式下，从并行输入来的数据被路由到 IC 的内部核心逻辑中，核心逻辑中的数据路由到输出管脚。在移位模式下，从前面单元中得到的串行数据被锁存在触发器 *Q*1 中，同时 *Q*1 中存储的数据移入到下一个边界扫描单元中。在 *Q*2 更新后，测试数据可以提供给内部逻辑或输出管脚。

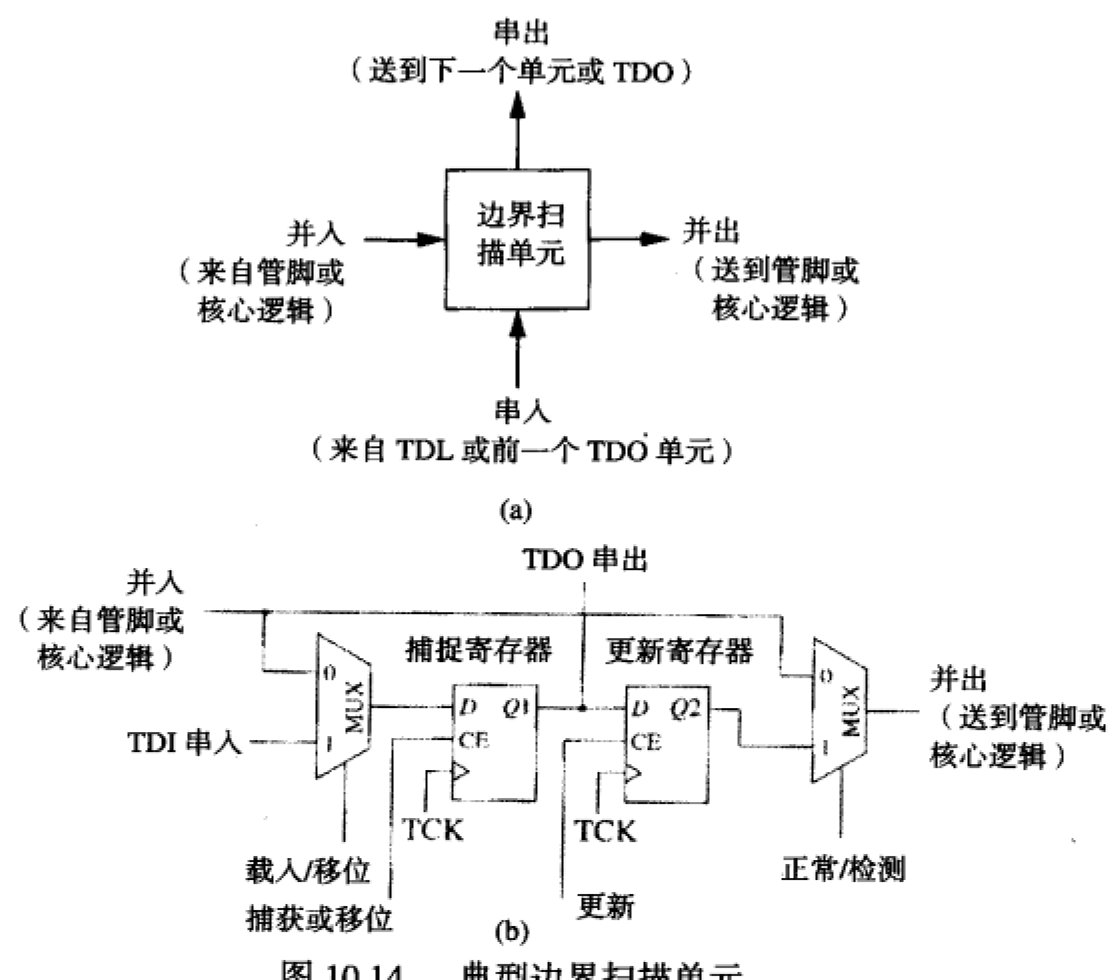


图 10.14　典型边界扫描单元

基本边界扫描的结构见图 10.15。在每个边界扫描 IC 的结构均如此。边界扫描寄存器被分为两部分：BSR1 和 BSR2。BSR1 表示移位寄存器，是由边界扫描单元中的 $Q1$ 触发器构成的。BSR2 表示 $Q2$ 触发器，当接收到更新信号时，它可以从 BSR1 中并行载入。串行输入设计(*TDI*)可以移入到寄存器中。每个 IC 上的 TAP 控制器包含一个状态机（参见图 10.16）。状态机的输入为 *TMS*，用 01 序列对 *TMS* 赋值，此 01 序列决定了是把 *TDI* 数据移入到指令寄存器中，还是移入到边界扫描单元。TAP 控制器和指令寄存器控制边界扫描单元的操作。

TAP 控制器状态机具有 16 个状态，其中状态 9 ~ 15 用于载入和更新指令寄存器，状态 2 ~ 8 用于载入和更新数据寄存器(BSR1)。TRST 信号如果被使用，则状态复位为测试逻辑复位状态。状态图具有一个有意思的属性，不论初始状态为何，只要 TMS 输入为 5 个 1 序列，则机器就复位并回到状态 0。

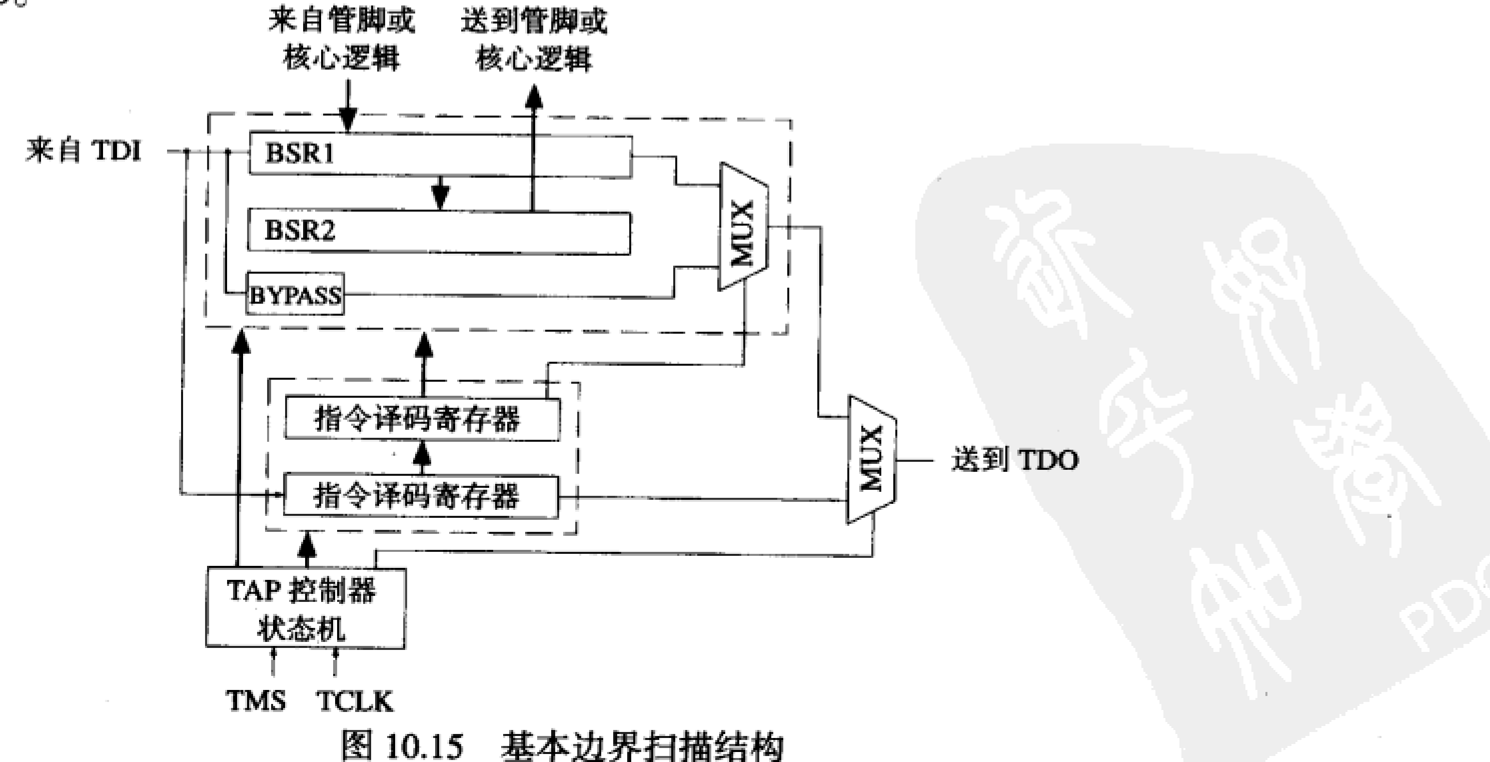


图 10.15　基本边界扫描结构

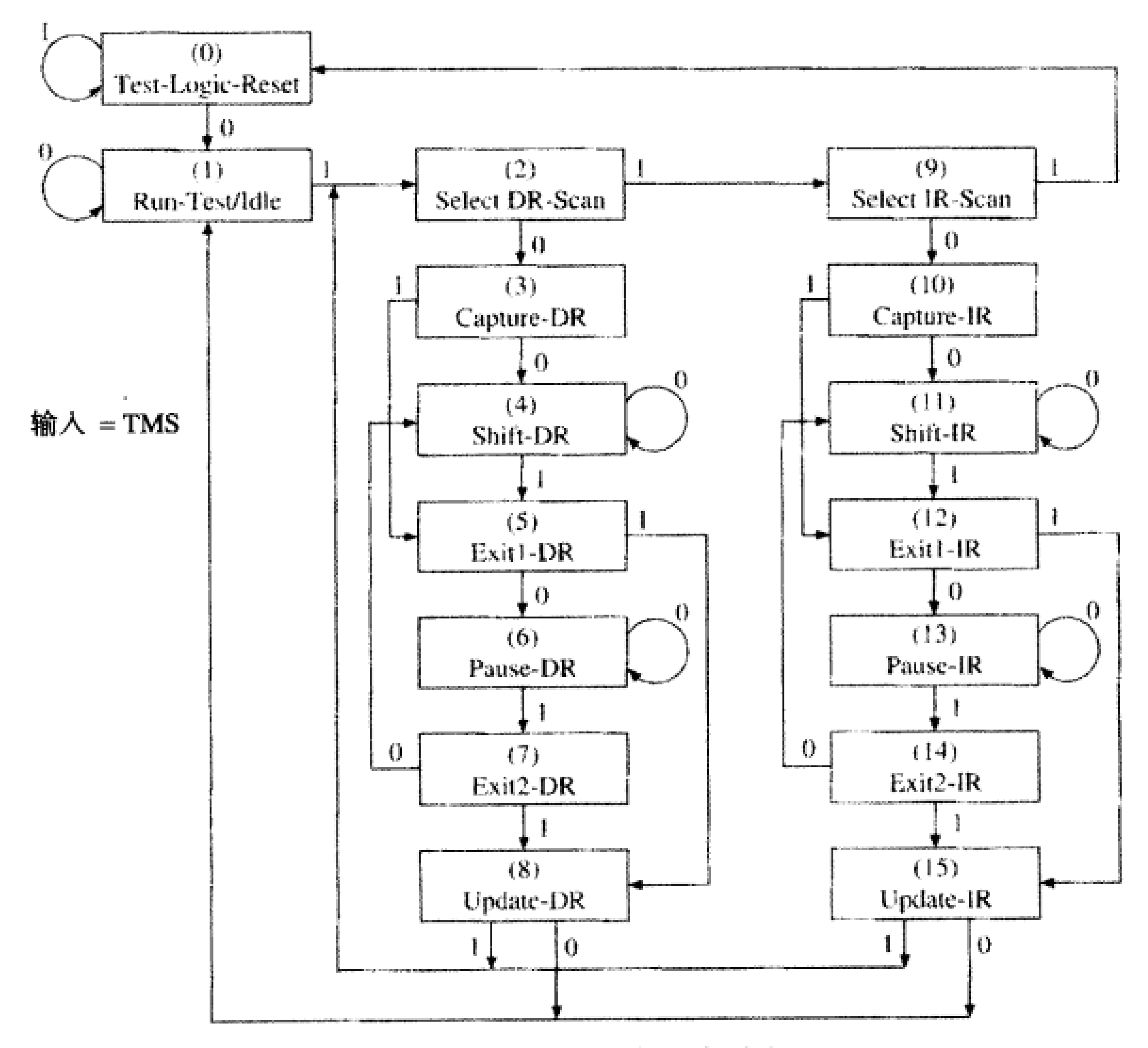


图 10.16 TAP 控制器状态机

IEEE 标准定义了如下指令：

- BYPASS: 此指令可以使 *TDI* 串行数据通过 IC 上一个 1 位旁路寄存器，而不是通过边界扫描寄存器。因此，当其他 IC 被测试时，PC 板上一个或多个 IC 都可以被旁路。
- SAMPLE/PRELOAD: 此指令可以通过核心逻辑的正常操作实现边界扫描寄存器的无干扰扫描。无论是把 IC 管脚上的数据传到核心逻辑，还是把核心逻辑中的数据传到 IC 管脚上，数据的传递都是无干扰的。此指令会对数据进行采样，并通过边界扫描寄存器进行扫描，测试数据被移入到 BSR 中。
- EXTEST: 此指令可以对电路板层面的互连测试，而且允许对不包含边界扫描测试特性的元件集群进行测试。测试数据被移入到 BSR 中，并随后被传输到输出管脚。从输入管脚得到的数据由 BSR 抓取。
- INTEST（可选）: 通过把测试数据移入到边界扫描寄存器中，此指令可以对核心逻辑进行测试。移入 BSR 的数据取代了从输入管脚中得到的数据，从核心逻辑中得到的输出数据被载入 BSR。
- RUNBIST（可选）: 此指令使 IC 中的内嵌自测试(BIST)逻辑开始执行（我们将在 10.5 节中介绍怎样用 BIST 生成测试序列和检查测试结果）。

还有一些其他的可选指令。用户自定义指令也可以使用。

IC 管脚、边界扫描寄存器和核心逻辑之间的数据路径是由执行的指令和 TAP 控制器的状态决定的。指令 Sample/Preload、Extest 和 Intest 的数据路径分别如图 10.17、图 10.18、图 10.19 中的粗线所示。在每种情况下，边界扫描寄存器 BSR1 和 BSR2 均被分为两部分：一部分与输入管脚相连，一部分与输出管脚相连。测试数据从 TDI 中被移入到 BSR1，并被移出到 TDO 中。

对于 Sample/Preload 指令（参见图 10.17），核心逻辑在正常模式下对从 IC 输入管脚中得到的输入进行操作，并输出到输出管脚上。当控制器在 CapureDR 状态，BSR1 从输入管脚和核心逻辑输出并行读入数据。在 UpdateDR 状态，BSR2 从 BSR1 中读取数据。

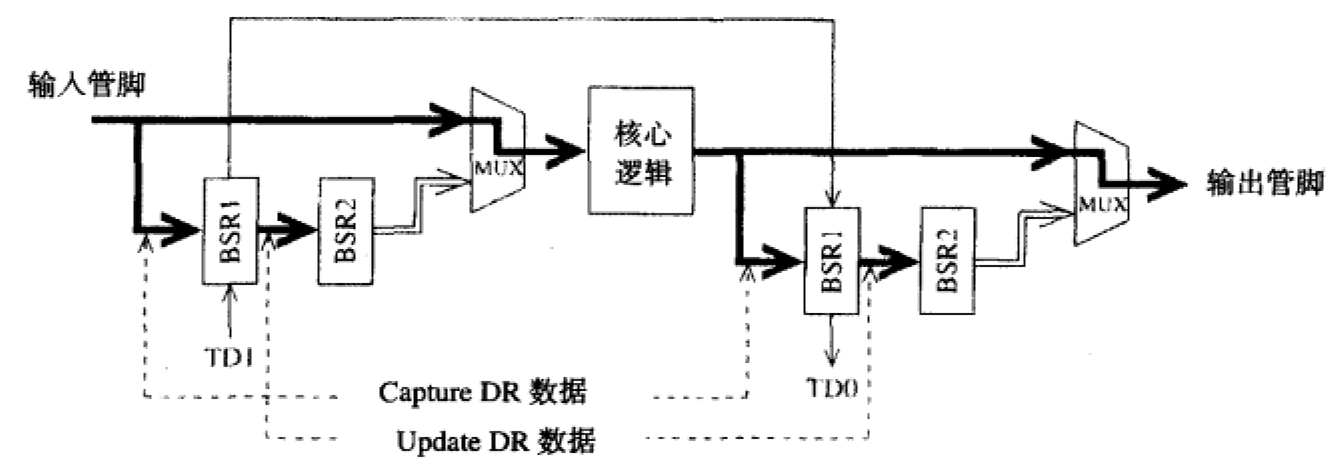


图 10.17 采样/前载指令的信号路径（粗线）

对于 Extest 指令（参见图 10.18），核心逻辑未被使用。在 UpdateDR 状态，BSR2 从 BSR1 中载入数据，而且数据被路由到 IC 的输出管脚。在 CapureDR 状态，输入管脚数据被载入到 BSR1 中。

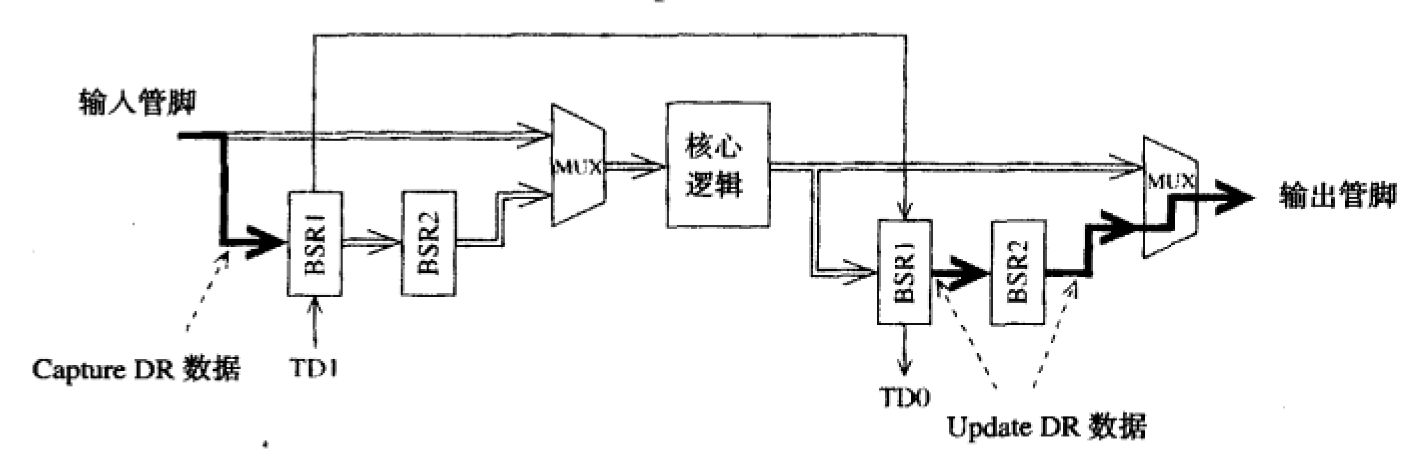


图 10.18 外部测试指令的信号路径（粗线）

对于 Intest 指令（参见图 10.19），IC 管脚未被使用。在 UpdateDR 状态，前面被移入到 BSR1 中的测试数据被载入到 BSR2 中，并路由到核心逻辑输入。在 CapureDR 状态，从核心逻辑中得到的数据被载入到 BSR1 中。

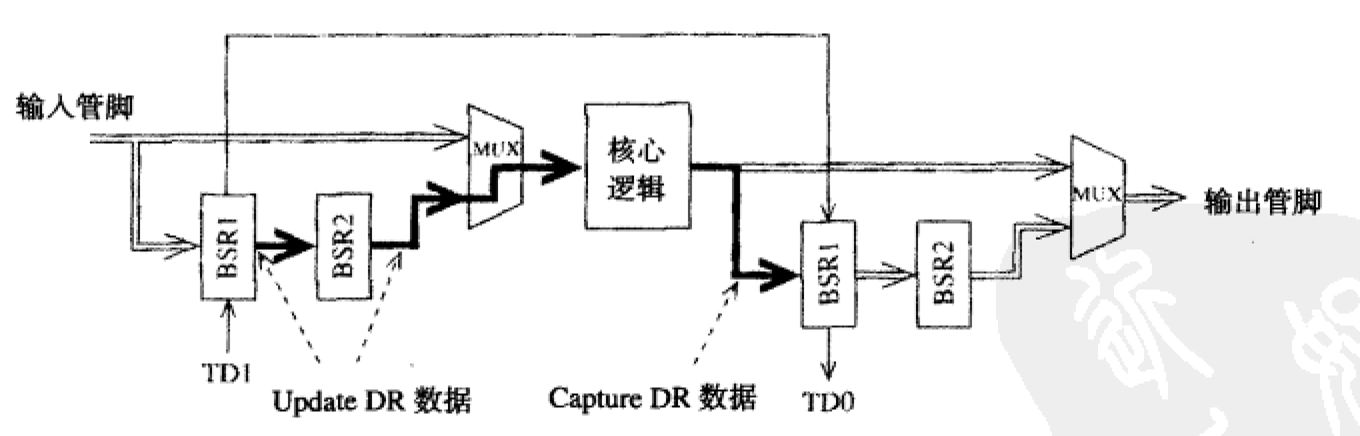


图 10.19 内部测试指令的信号路径（粗线）

下面我们介绍几个简单的例子，以说明两个 IC 间的何种连接可以通过 Sample/Preload 和 Extest 指令进行测试。我们要对 PC 板线路的开路和短路进行测试。两个 IC 均含有 2 个输入管脚和 2 个输出管脚，如图 10.20 所示。测试数据通过 *TDI* 被移入到 BSR 中，然后从输入管脚中得到的数据并行载入到两个 BSR 中，并通过 *TDO* 移出。假设每个 IC 上的指令寄存器的长度均为 3

位，而且 EXTEST 编码为 000，SAMPLE/PRELOAD 编码为 001。IC1 中的核心逻辑是一个时钟振荡器，它由一个反相器连接的两个触发器构成。IC2 中的核心逻辑是由一个反相器和 XOR 门构成的。这两个 IC 连接起来形成一个 2 位计数器。

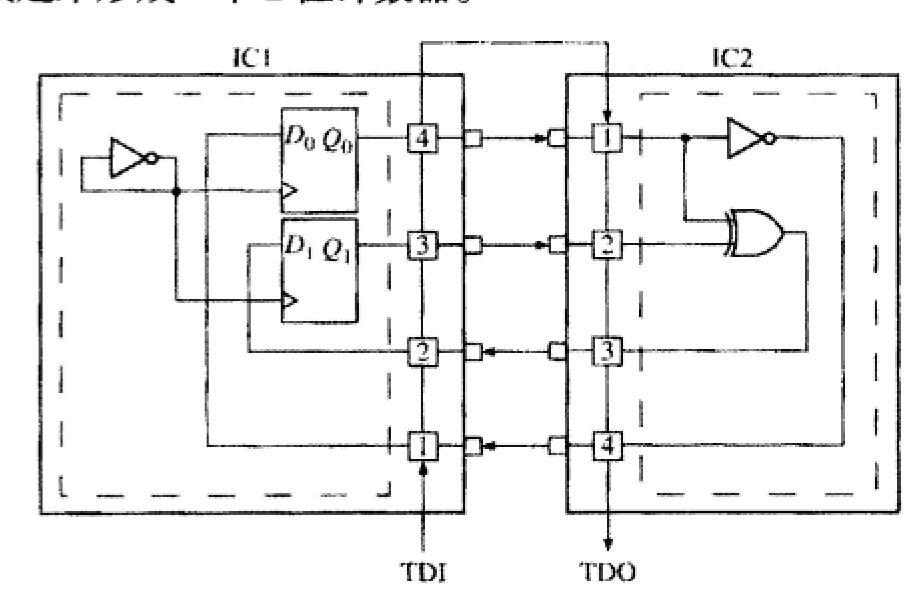


图 10.20　使用边界扫描测试互连

对 IC 之间连接的测试需要如下步骤：

1. 通过在 *TMS* 中输入序列“11111”，把 TAP 状态机复位为 Test-Logic-Reset 状态。
2. 对于两个 IC，在用 Sample/Preload 指令扫描时，使用如下 *TMS* 和 *TDI* 序列（参见图 10.16）：

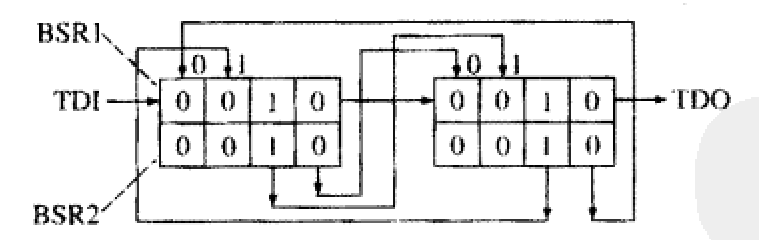
```
State:  0  1  2  9  10  11  11  11  11  11  11  12  15  2
TMS:    0  1  1  0  0   0   0   0   0   0   1   1   1
TDI:    -  -  -  -  -   1   0   0   1   0   0   -   -
```

   TMS 序列 01100 使 TAP 控制器转为 Shift-IR 状态。在此状态中，Sample/Preload 指令（编码为 001）的复制指令分别被移入到两个 IC 的指令寄存器中。在 Update _IR 状态，指令被载入到指令译码寄存器中。随后，TAP 控制器变回到 Select DR-scan 状态。

3. 把测试数据的第一集合提前载入到 IC 中。使用的 *TMS* 和 *TDI* 序列如下：

```
State:  2  3  4  4  4  4  4  4  4  4  5  8  2
TMS:    0  0  0  0  0  0  0  0  0  1  1  1
TDI:    -  -  0  1  0  0  0  1  0  0  -  -
```

   在 Shift-DR 状态，数据被移入到 BSR1 中，并且在 Update _DR 状态被转移到 BSR2 中，结果如下所示：



4. 对于两个 IC，在用 EXTEST 指令扫描时，使用如下序列：

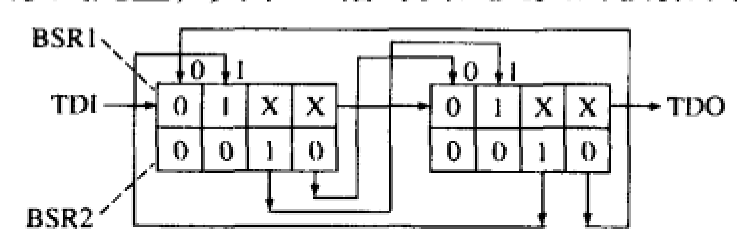
```
State:  2  9  10  11  11  11  11  11  11  12  15  2
TMS:    1  0  0   0   0   0   0   0   1   1   1
TDI:    -  -  -   0   0   0   0   0   0   -   -
```

   在 Shift-IR 状态，EXTEST 指令(000)被扫描到指令寄存器中，并在 Update-IR 状态被载入到指令译码器中。此时，提前载入的测试数据输出到输出管脚，并通过印制电路板线路传播到两个 IC 的相邻输入管脚。

5. 从 IC 输入中抓取测试结果。把此数据扫描输出到 *TDO*，并且使用下列序列对第二个测试数据集进行扫描。

```
State:  2  3  4  4  4  4  4  4  4  4  5  8  2
TMS:    0  0  0  0  0  0  0  0  0  1  1  1
TDI:    -  -  1  0  0  0  1  0  0  0  -  -
TDO:    -  -  x  x  1  0  x  x  1  0  -  -
```

在 Capture-DR 状态，输入管脚的数据被载入到 BSR1 中。此时，如果检测到任何故障，则 BSR 应该按如下形式配置，其中 X 指与测试无关的被抓取数据。



当新的测试数据被移入且在 Shift-DR 状态，测试结果从 BSR1 中被移出。在 Update-IR 状态，新数据被载入到 BSR2 中。

6. 从 IC 输入中抓取测试结果。把此数据扫描输出到 TDO，并且使用下列序列对所有的 0 进行扫描。

```
State:  2  3  4  4  4  4  4  4  4  4  5  8  2  9  0
TMS:    0  0  0  0  0  0  0  0  0  1  1  1  1  1
TDI:    -  -  0  0  0  0  0  0  0  0  -  -  -  -
TDO:    -  -  x  x  0  1  x  x  0  1  -  -  -  -
```

在 Capture-DR 状态，输入管脚的数据被载入到 BSR1 中。当所有的 0 被移入且在 Shift-DR 状态，测试结果从 BSR1 中被移出。在 Update-IR 状态，所有的 0 被载入到 BSR2 中。控制器返回到 Test-Logic-Reset 状态，IC 正常工作。如果观察到 *TDO* 序列与上面给定的序列相符，则互连测试通过。

图 10.15 的所示基本边界扫描结构的 VHDL 代码示于图 10.21。程序中只有三个主要指令（EXTEST, SAMPLE/PRELOAD 和 BYPASS），均通过 3 位指令寄存器实现。这些指令分别编码为 000, 001 和 111。BSR 中单元的个数用一个类属变量表示。第二个类属变量（CellType，且为位矢量数据类型）用来设定每个单元是输入单元还是输出单元。case 语句实现 TAP 控制器状态机。在状态 Capture-IR, Shift-IR 和 Update-IR 中，指令代码被扫描和载入 *IDR* 中，并在状态 Capture-DR、Shift-DR 和 Update-DR 中指令开始执行。这些状态中的操作取决于被执行的指令。寄存器的更新和状态的改变均在 *TCK* 上升沿发生。此 VHDL 代码实现了 IEEE 边界扫描标准要求的大部分功能，但是并没有完全遵守 IEEE 标准。

```
-- VHDL for Boundary Scan Architecture of Figure 10-15

entity BS_arch is
  generic(NCELLS: natural range 2 to 120 := 2);
          -- number of boundary scan cells
  port(TCK, TMS, TDI: in bit;
       TDO: out bit;
       BSRin: in bit_vector(1 to NCELLS);
       BSRout: inout bit_vector(1 to NCELLS);
       CellType: bit_vector(1 to NCELLS));
       -- '0' for input cell, '1' for output cell
end BS_arch;

architecture behavior of BS_arch is
  signal IR, IDR: bit_vector(1 to 3);          -- instruction registers
  signal BSR1, BSR2: bit_vector(1 to NCELLS); -- boundary scan cells
  signal BYPASS: bit;                          -- bypass bit
  type TAPstate is (TestLogicReset, RunTest_Idle,
```

图 10.21　基本边界扫描结构的 VHDL 程序

```
    SelectDRScan, CaptureDR, ShiftDR, Exit1DR, PauseDR, Exit2DR, UpdateDR,
    SelectIRScan, CaptureIR, ShiftIR, Exit1IR, PauseIR, Exit2IR, UpdateIR);
  signal St: TAPstate;                          -- TAP Controller State
begin
  process (TCK)
  begin
    if TCK'event and TCK='1' then
     -- TAP Controller State Machine
      case St is
        when TestLogicReset =>
          if TMS='0' then St <= RunTest_Idle; else St<=TestLogicReset; end if;
        when RunTest_Idle =>
          if TMS='0' then St <= RunTest_Idle; else St <= SelectDRScan; end if;
        when SelectDRScan =>
          if TMS='0' then St <= CaptureDR; else St <= SelectIRScan; end if;
        when CaptureDR =>
          if IDR = "111" then BYPASS <= '0';
          elsif IDR = "000" then  -- EXTEST (input cells capture pin data)
           BSR1 <= (not CellType and BSRin) or (CellType and BSR1);
         elsif IDR = "001" then - SAMPLE/PRELOAD
           BSR1 <= BSRin;
         end if;  -- all cells capture cell input data
         if TMS='0' then St <= ShiftDR; else St <= Exit1DR; end if;
       when ShiftDR =>
         if IDR = "111" then BYPASS <= TDI; -- shift data through bypass reg.
         else BSR1 <= TDI & BSR1(1 to NCELLS-1); end if;
          -- shift data into BSR
         if TMS='0' then St <= ShiftDR; else St <= Exit1DR; end if;
       when Exit1DR =>
         if TMS='0' then St <= PauseDR; else St <= UpdateDR; end if;
       when PauseDR =>
         if TMS='0' then St <= PauseDR; else St <= Exit2DR; end if;
       when Exit2DR =>
         if TMS='0' then St <= ShiftDR; else St <= UpdateDR; end if;
       when UpdateDR =>
         if IDR = "000" then -- EXTEST (update output reg. for output cells)
           BSR2 <= (CellType and BSR1) or (not CellType and BSR2);
         elsif IDR = "001" then     -- SAMPLE/PRELOAD
           BSR2 <= BSR1;            -- update output reg. in all cells
         end if;
         if TMS='0' then St <= RunTest_Idle; else St <= SelectDRScan; end if;
       when SelectIRScan =>
         if TMS='0' then St <= CaptureIR; else St <= TestLogicReset; end if;
       when CaptureIR =>
         IR <= "001"; -- load 2 LSBs of IR with 01 as required by the standard
         if TMS='0' then St <= ShiftIR; else St <= Exit1IR; end if;
       when ShiftIR =>
         IR <= TDI & IR(1 to 2);  -- shift in instruction code
         if TMS='0' then St <= ShiftIR; else St <= Exit1IR; end if;
       when Exit1IR =>
         if TMS='0' then St <= PauseIR; else St <= UpdateIR; end if;
       when PauseIR =>
         if TMS='0' then St <= PauseIR; else St <= Exit2IR; end if;
       when Exit2IR =>
         if TMS='0' then St <= ShiftIR; else St <= UpdateIR; end if;
       when UpdateIR =>
         IDR <= IR;   -- update instruction decode register
         if TMS='0' then St <= RunTest_Idle; else St <= SelectDRScan; end if;
      end case;
    end if;
  end process;

  TDO <= BYPASS when St = ShiftDR and IDR = "111"  -- BYPASS
    else BSR1(NCELLS) when St=ShiftDR   -- EXTEST or SAMPLE/PRELOAD
    else IR(3) when St=ShiftIR;

  BSRout <= BSRin when (St = TestLogicReset or not (IDR = "000"))
    else BSR2;                          -- define cell outputs
end behavior;
```

图 10.21（续） 基本边界扫描结构的 VHDL 代码

图 10.20 所示互连测试的 VHDL 代码示于图 10.22。*TMS* 和 *TDI* 测试模式是步骤 2 到步骤 6 中测试模式的拼接。IC1 和 IC2 均为基本边界扫描结构。每个 IC 的外部连接和内部逻辑都进行了

具体设定。内部时钟频率是随意选定的，只要与测试时钟频率不同即可。测试进程先运行内部逻辑，接着运行扫描测试，然后再运行内部逻辑。测试结果可以验证 IC 逻辑运行是否正确，扫描测试是否能得到预计的结果。

```
-- Boundary Scan Tester

entity system is
end system;

architecture IC_test of system is
  component BS_arch is
    generic(NCELLS:natural range 2 to 120 := 4);
    port(TCK, TMS, TDI: in bit;
         TDO: out bit;
         BSRin: in bit_vector(1 to NCELLS);
         BSRout: inout bit_vector(1 to NCELLS);
         CellType: in bit_vector(1 to NCELLS));
         -- '0' for input cell, '1' for output cell
  end component;

  signal TCK, TMS, TDI, TDO, TDO1: bit;
  signal Q0, Q1, CLK1: bit;
  signal BSR1in, BSR1out, BSR2in, BSR2out: bit_vector(1 to 4);
  signal count: integer := 0;

  constant TMSpattern: bit_vector(0 to 62) :=
    "011000000011100000000011110000000111000000000111000000000111111";
  constant TDIpattern: bit_vector(0 to 62) :=
    "000001001000000010001000000000000000001000100000000000000000000";
begin
  BS1: BS_arch port map(TCK, TMS, TDI, TDO1, BSR1in, BSR1out, "0011");
  BS2: BS_arch port map(TCK, TMS, TDO1, TDO, BSR2in, BSR2out, "0011");
  -- each BSR has two input cells and two output cells
  BSR1in(1) <= BSR2out(4);                -- IC1 external connections
  SR1in(2) <= BSR2out(3);
  BSR1in(3) <= Q1;                        -- IC1 internal logic
  BSR1in(4) <= Q0;
  CLK1 <= not CLK1 after 7 ns;             -- internal clock
  process(CLK1)
  begin
    if CLK1 = '1' then                      -- D flip-flops
      Q0 <= BSR1out(1);
      Q1 <= BSR1out(2);
    end if;
  end process;

  BSR2in(1) <= BSR1out(4);                 -- IC2 external connections
  BSR2in(2) <= BSR1out(3);
  BSR2in(3) <= BSR2out(1) xor BSR2out(2);  -- IC2 internal logic
  BSR2in(4) <= not BSR2out(1);

  TCK <= not TCK after 5 ns;                        -- test clock

  process
  begin
    TMS <= '1';
    wait for 70 ns;                       -- run internal logic
    wait until TCK = '1';
    for i in TMSpattern'range loop        -- run scan test
      TMS <= TMSpattern(i);
      TDI <= TDIpattern(i);
      wait for 0 ns;
      count <= i + 1;                       -- count triggers listing output
      wait until TCK = '1';
    end loop;
    wait for 70 ns;                       -- run internal logic
    wait;                                 -- stop
  end process;
end IC_test;
```

图 10.22　互连测试实例 VHDL 程序

10.5 内嵌自测试

随着数字系统变得越来越复杂，数字系统的测试也变得越来越难，费用越来越昂贵。此问题的一个解决方案为：在 IC 上添加逻辑电路，这样它就可以进行自测试。这一方法称为内嵌自测试，或 BIST，其基本用法示于图 10.23。用一个片上测试生成器对测试中的电路加载测试模式。测试结果输出到相应的监视器上，如果检测到错误的输出模式，则输出错误信号。

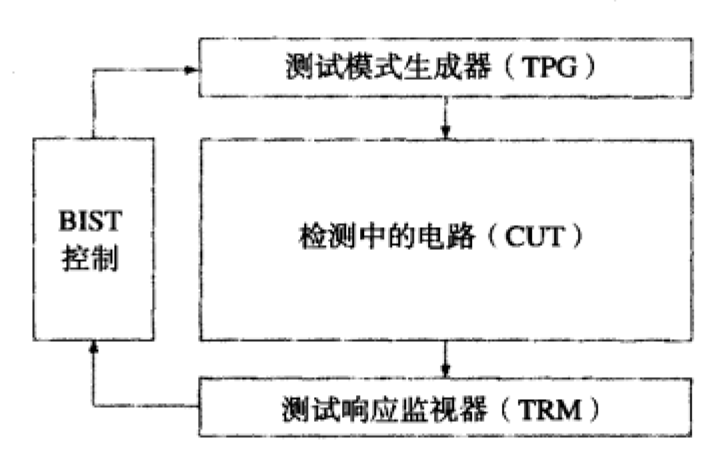


图 10.23 通用 BIST 结构

BIST 常用来测试内存。内存芯片的常规结构使其易于生成测试模式。一个 RAM 的自测试平台框图如图 10.24 所示。BIST 控制器首先激活写数据生成器和地址计数器，所以数据可以写入到 RAM 的任何存储单元中。随后激活地址计数器和读数据生成器，所以从每个 RAM 存储单元中读取的数据可以跟读数据生成器的输出进行比较，并验证其正确性。内存通常用“棋盘模式”进行测试。“棋盘模式”是指在所有内存单元中写入交替出现的 0 和 1，然后把它们读出。例如，我们可以首先在内存的偶地址中写入交替出现的 0 和 1，再在奇地址中写入交替出现的 1 和 0。把它们全部读回后，再把奇偶地址的存储模式互换以完成测试。还有另外一种测试方法：March 测试。首先读取每个单元中的数据，对其取补后再写回原先的存储单元中。一直持续此过程直到所有的内存数组被遍历。然后按照与上次地址相反的顺序重新进行上面的操作。

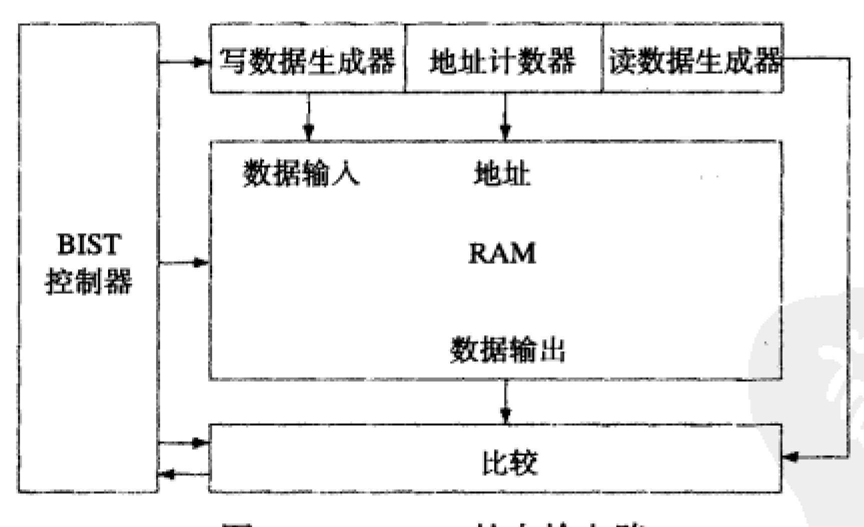


图 10.24 RAM 的自检电路

为了简化测试电路，我们使用了签名寄存器。签名寄存器把输出数据压缩为一个较短的比特数据串，称为签名。我们把该签名同功能正确的元件的签名进行比较。多输入签名寄存器(MISR)把多个输出流进行联合并压缩为单一签名。图 10.25 给出了一个 RAM 自测试电路的简单模型。读数据生成器和比较器被 MISR 所取代。一种类型的 MISR 是通过把 RAM 中存储的所有数据字节都加起来并形成一个测试和。在对 ROM 进行测试时，由于无须写数据生成器，所以图 10.25 可以进一步化简。

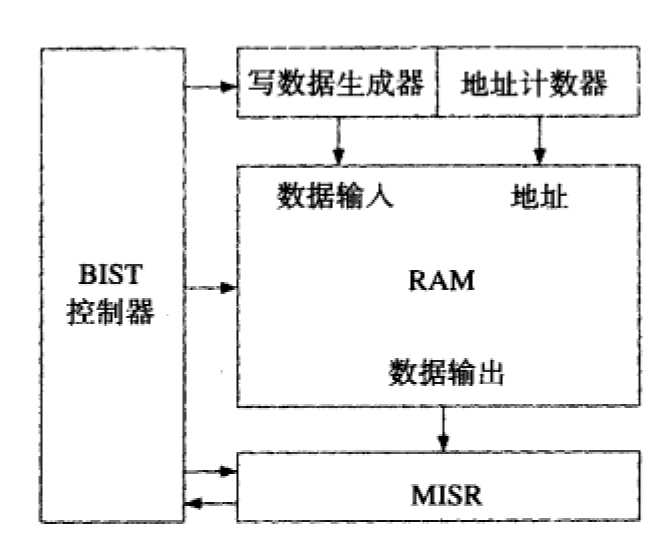


图 10.25　具有特征寄存器的 RAM 的自检电路

线性反馈移位寄存器(LFSR)通常用于生成测试模式和把测试输出压缩成签名。LFSR 是一个移位寄存器，其串行输入位同当前移位寄存器中的某些位呈线性关系。影响串行输入的位置比特称为抽头。通常来说，LFSR 由两个或两个以上的触发器构成，触发器的输出在取 XOR 后反馈回第一个触发器。由于异或操作等效于模 2 加，且加法为线性操作，所以称为“线性”。图 10.26 是一个 LFSR 的实例。图中第一个和第四个触发器的输出经过 XOR 操作后反馈回第一个触发器，所以抽头位置为 1 和 4。

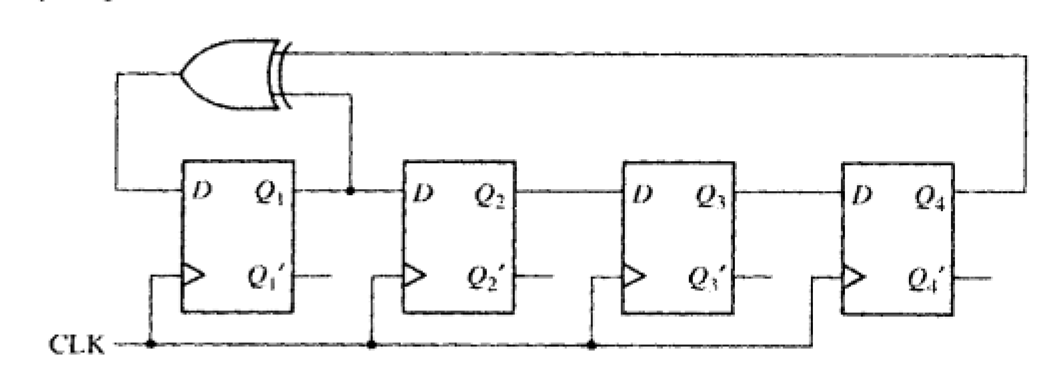


图 10.26　4 位线性反馈移位寄存器（LFSR）

通过恰当地选择输出进行 XOR 运算并反馈回第一个触发器，我们可以用一个 n 位的移位寄存器生成 2^n-1 个不同的数据串。除了全 0 以外，所有可能的数据串都可以生成。由图 10.26 的 LFSR 生成的数据串为

1000，1100，1110，1111，0111，1011，0101，1010，1101，0110，0011，

1001，0100，0010，0001，1000，...

这些数据串没有明确的排列方式，是随机排列的。这样的 LFSR 通常称为伪随机模式生成器或 PRPG。显然，由于 PRPG 使用很少的硬件电路生成了大量的测试数据串，所以其在 BIST 中是很有用的。某个 LFSR 的长度 $n=4\sim32$，它可以生成 2^n-1 个不同的数据串，其反馈组合如表 10.4 所示。

表 10.4　最大长度 LFSR 序列的反馈

n	反馈
4, 6, 7	$Q_1 \oplus Q_n$
5	$Q_2 \oplus Q_5$
8	$Q_2 \oplus Q_3 \oplus Q_4 \oplus Q_8$
12	$Q_1 \oplus Q_4 \oplus Q_6 \oplus Q_{12}$
14, 16	$Q_3 \oplus Q_4 \oplus Q_5 \oplus Q_n$
24	$Q_1 \oplus Q_2 \oplus Q_7 \oplus Q_{24}$
32	$Q_1 \oplus Q_2 \oplus Q_{22} \oplus Q_{32}$

如果需要全 0 测试模式，则可以在一个 n 位 LFSR 上添加一个具有 n–1 个输入的 AND 门，当 n = 4 时，改后的逻辑电路图示于图 10.27。在状态 0001 时，下一状态即为 0000；当前状态为 0000 时，下一状态为 1000；否则序列与图 10.26 相同。

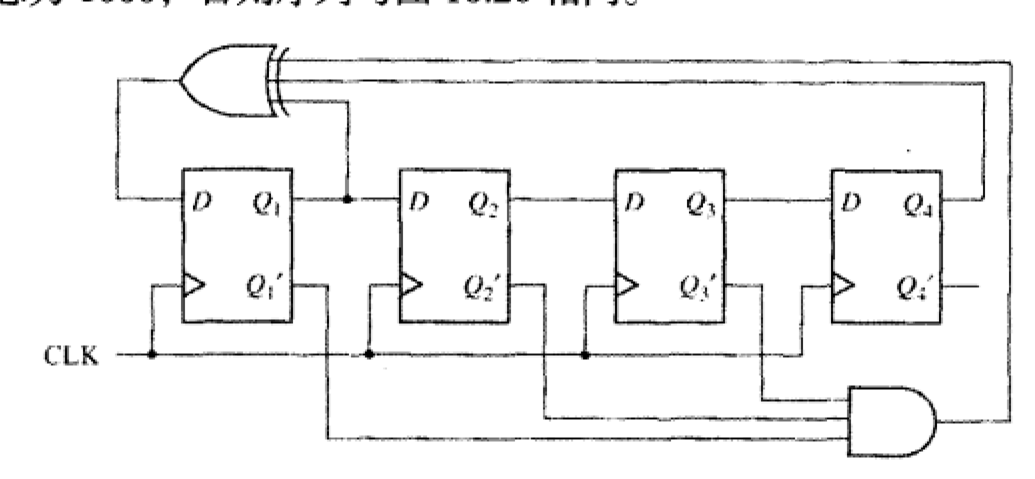


图 10.27 具有 0000 状态的改进的 LFSR

通过在 LFSR 上添加 XOR 门，我们可以构建一个 MISR，如图 10.28 所示。测试数据($Z_1Z_2Z_3Z_4$)取 XOR 后，与每个时钟一起进入寄存器，最终生成一个签名，并将其与已知正确功能元件的签名进行比较。这种签名分析可以检测到很多错误，但是不能检测到所有的错误。一个 n 位签名寄存器把所有可能输入串映射到 2^n 个可能签名中的一个。其中一个签名是正确的，而其他签名表明了错误的出现。错误输入序列映射到正确签名的可能性为 $1/2^n$。

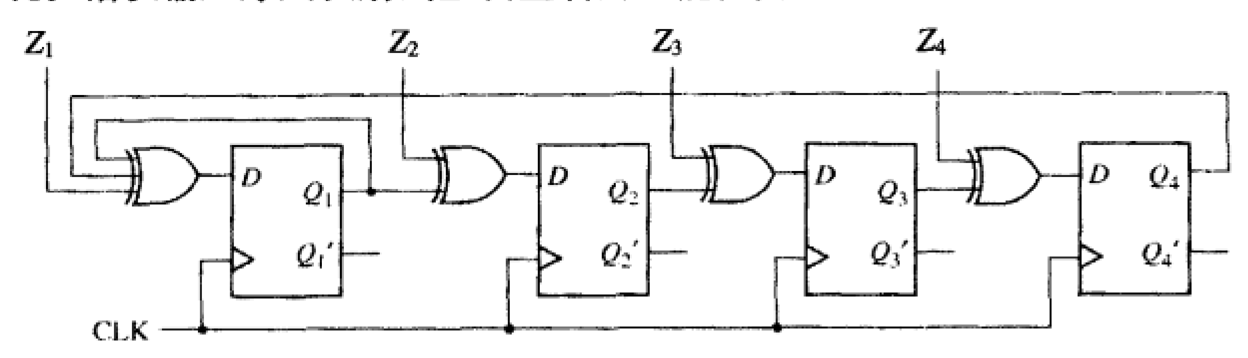


图 10.28 多输入签名寄存器(MISR)

对于图 10.28 中的 MISR，假设正确的输入序列为 1010, 0001, 1110, 1111, 0100, 1011, 1001, 1000, 0101, 0110, 0011, 1101, 0111, 0010, 1100。假设 MISR 的初始内容为 0000，则这些序列映射的签名为 0010。只要输入序列中有任何一位不同，则其对应的签名也将不同。例如，如果序列中的 0001 变为 1001，则最终序列映射的签名为 0000。大多数具有两个错误的序列可以被检测出来，但是如果我们把原始序列中的 0001 变为 1001，0010 变为 0110，则对应的签名为 0010，签名是正确的，但是错误并没有被检测出来。

我们已经对 BIST 的多个结构类型进行了介绍，在这些结构类型中，有两种结构很流行：STUMPS 结构和 BILBO 结构。

STUMPS 是指使用 MISR 和并行 SRSG 进行自测试(Self-Testing Using an MISR and Parallel SRSG)。SRSG 是指移位寄存器序列生成器。STUMPS 是一种使用扫描链的 BIST 结构。STUMPS 结构的简单框图如图 10.29 所示。伪随机序列生成器把测试激励馈回扫描链，在一个捕捉周期后，测试响应分析器接收到测试响应。STUMPS 的测试步骤如下所示：

1. 从测试模式生成器(LFSR)中取出数据串，并用所有的扫描链进行扫描。
2. 变为普通功能模式，并使用一次系统时钟。
3. 移出扫描链到测试响应分析器(MISR)，生成测试特征位。

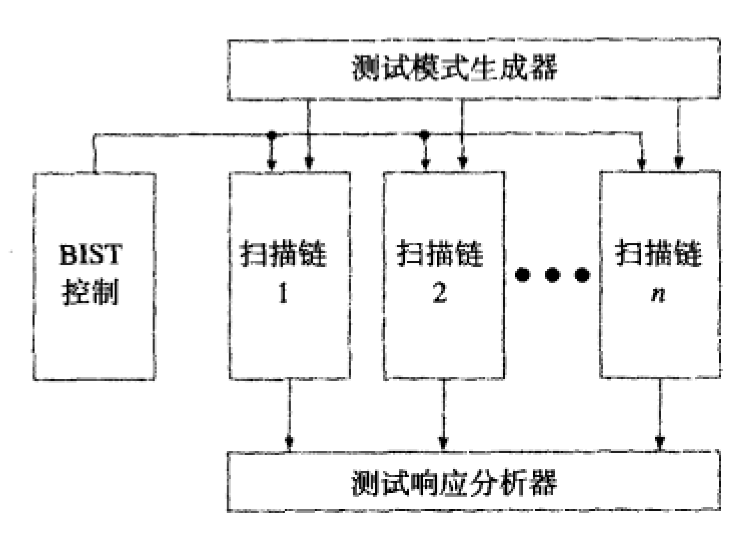


图 10.29　STUMPS 结构

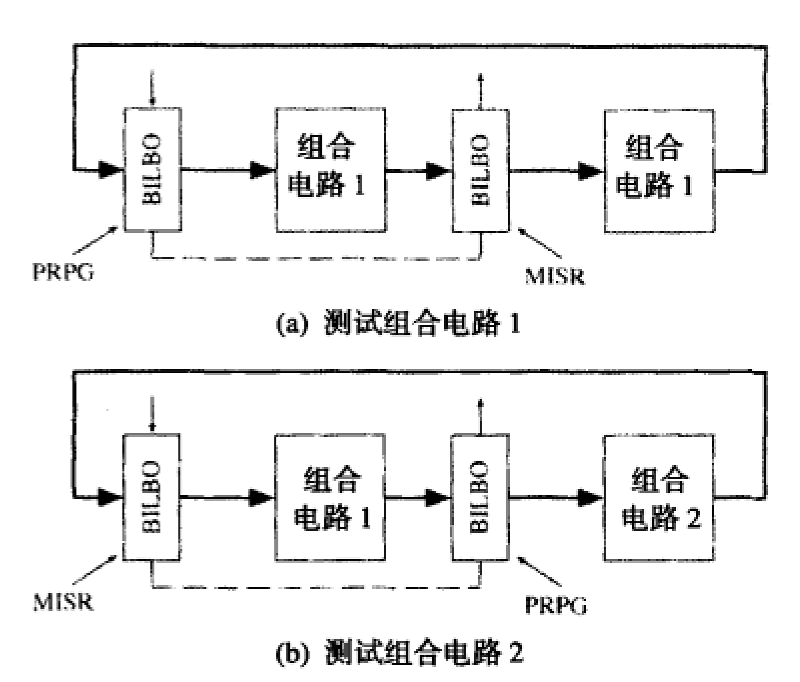
如果扫描链有 100 个扫描单元，则步骤 1 和步骤 3 将使用 100 个时钟才可完成。所有的扫描链首先应该由伪随机序列生成器赋值；因此，较长的扫描链就需要较长的测试时间。由于**每次扫描可以完成一次测试**，所以我们应该使用数量巨大的并行扫描链。由于扫描链可以并行加载，所以使用并行扫描链可以减少赋值时间。

STUMPS 结构最初是从多芯片模块中发展起来的[7]。每个逻辑芯片的扫描链都可以并行载入伪随机序列。所需的时钟周期数量等于最长的扫描链中触发器的个数。如果在最长的扫描链中有 m 个扫描单元，则实现一次测试将需要 $2m+1$ 个时钟循环（m 个循环用于扫描进入，1 个用于抓取，另外 m 个用于扫描输出）。较短的扫描链将溢出到 MISR 中，但是这并不影响最终签名的正确性。

为了减少测试时间，步骤 1 和步骤 3 可以合二为一。当扫描链在一次测试后，未被载入到 MISR 时，下一个伪随机序列将同时被载入到扫描链中（例如，当测试 I 的测试响应正在被移出时，测试 I + 1 的测试串就可以移入）。假设把当前测试的扫描输出和下一次测试的扫描输入合并，则每个测试矢量都将消耗 $m+1$ 个时钟循环，对于 n 个测试矢量来说，就需要 $n(m+1)+m$ 个时钟循环。注意，最后的 m 个时钟循环用于实现最后一个扫描输出。

与刚刚讨论的结构（每次扫描进行一次测试）不同，每个时钟进行一次测试的结构将用于快速测试。此类结构被称为 BILBO（内嵌逻辑模块观察器，Built-In Logic Block Observer）技术。在 BILBO 结构中，我们对扫描寄存器进行修改，把扫描寄存器的一部分用做状态寄存器、数据串生成器、签名寄存器或移位寄存器。当用做移位寄存器时，测试数据按通常的方法进行扫描输入和输出。在测试时，扫描寄存器的一部分可以被用做测试数据串生成器(PRPG)，还有一部分作为签名寄存器(MISR)对组合模块中的一个进行测试。当测试结束时，扫描寄存器变为正常工作的状态寄存器模式。在 BILBO 寄存器初始化完毕后，由于在扫描链中测试数据串还未被载入，所以在每个时钟循环都可以进行一次测试。因此，我们可以把此结构分类为每个时钟循环进行一次测试的 BILBO 结构。BILBO 的测试长度较短，但是对测试硬件需要较高。

用两个组合模块进行电路测试的 BILBO 寄存器的布线如图 10.30 所示。当第一个 BILBO 用做 PRPG、第二个用做 MISR 时，组合电路 1 被测试。如果要对组合电路 2 进行测试，则只需把 BILBO 的用途对调。在正常工作模式下，两个 BILBO 都用做相关组合逻辑电路的寄存器。在进行扫描输入输出时，两个 BILBO 都处于移位寄存器工作模式。



(a) 测试组合电路 1

(b) 测试组合电路 2

图 10.30　使用 BILBO 寄存器的 BIST

图 10.31 展示了一个 4 位 BILBO 寄存器的结构。控制输入 B_1 和 B_2 决定工作模式。Si 和 So 是移位寄存器模式下的串行输入输出。Z 是组合逻辑电路的输入。此 BILBO 寄存器的逻辑等式为

$$D_1 = Z_1B_1 \oplus (Si\ B_2' + FB\ B_2)(B_1' + B_2)$$
$$D_i = Z_iB_1 \oplus Q_{i-1}(B_1' + B_2) \quad (i > 1)$$

当 $B_1 = B_2 = 0$ 时，这些等式可以化简为

$$D_1 = Si \text{ 和 } D_i = Q_{i-1} \quad (i > 1)$$

上式与移位寄存器的模式相对应。当 $B_1 = 0$ 且 $B_2 = 1$ 时，等式可以化简为

$$D_1 = FB, \quad D_i = Q_{i-1}$$

上式与 PRPG 模式相对应，其 BILBO 寄存器与图 10.26 中的寄存器等效。当 $B_1 = 1$ 且 $B_2 = 0$ 时，等式化简为

$$D_1 = Z_1, \quad D_i = Z_i$$

上式与一般操作模式相对应。当 $B_1 = B_2 = 1$ 时，等式可以化简为

$$D_1 = Z_1 \oplus FB, \qquad D_i = Z_i \oplus Q_{i-1}$$

上式与 MISR 模式相对应，其 BILBO 寄存器与图 10.28 中的寄存器等效。总之，BILBO 操作模式如下所示：

B1B2	操作模式
00	移位寄存器
01	PRPG
10	正常
11	MISR

图 10.32 展示了 n 位 BILBO 寄存器的 VHDL 代码。NBITS 是一个大小范围在 4 ~ 8 的 generic 参数，其值等于比特个数。除了添加了一个时钟使能信号(CE)以外，此寄存器的功能与图 10.31 所示寄存器的功能相同。LFSR 的反馈(FB)取决于比特数。

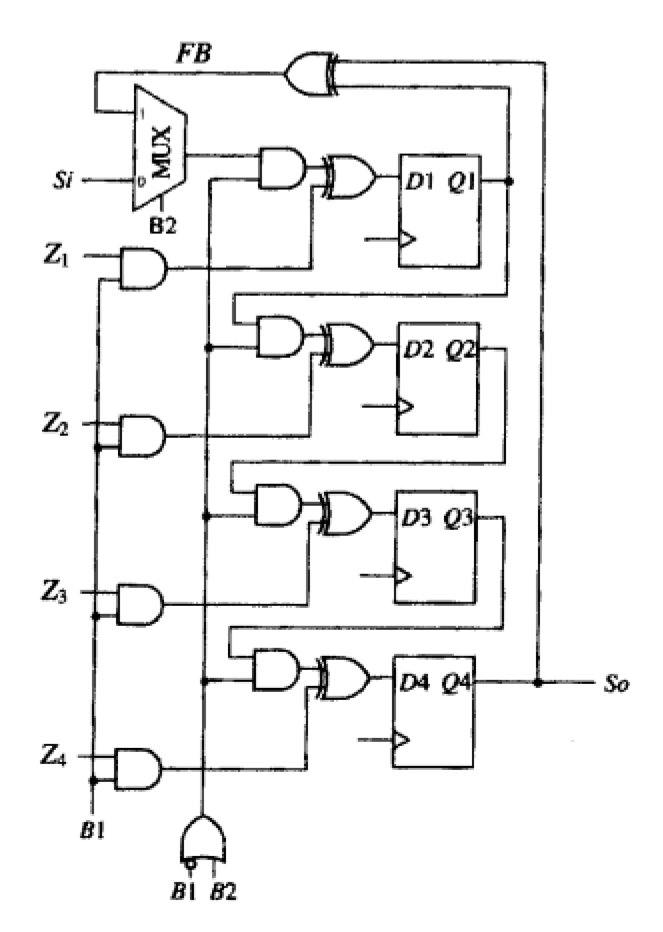


图 10.31　4 位 BILBO 寄存器

```
entity BILBO is                -- BILBO Register
  generic (NBITS: natural range 4 to 8 := 4);
  port (Clk, CE, B1, B2, Si: in bit;
        So: out bit;
        Z: in bit_vector(1 to NBITS);
        Q: inout bit_vector(1 to NBITS));
end BILBO;

architecture behavior of BILBO is
  signal FB: bit;
begin
  Gen8: if NBITS = 8 generate
    FB <= Q(2) xor Q(3) xor Q(NBITS); end generate;
  Gen5: if NBITS = 5 generate
    FB <= Q(2) xor Q(NBITS); end generate;
  GenX: if not(NBITS = 5 or NBITS = 8) generate
    FB <= Q(1) xor Q(NBITS); end generate;
  process(Clk)
    variable mode: bit_vector(1 downto 0);
  begin
    if (Clk = '1' and CE = '1') then
      mode := B1 & B2;
      case mode is
        when "00" =>      -- Shift register mode
          Q <= Si & Q(1 to NBITS-1);
        when "01" =>      -- Pseudo Random Pattern Generator mode
          Q <= FB & Q(1 to NBITS-1);
        when "10" =>      -- Normal Operating mode
          Q <= Z;
        when "11" =>      -- Multiple Input Signature Register mode
          Q <= Z(1 to NBITS) xor (FB & Q(1 to NBITS-1));
      end case;
    end if;
  end process;
  So <= Q(NBITS);
end behavior;
```

图 10.32　图 10.31 中 BILBO 寄存器的 VHDL 代码

图 10.33 所示系统中使用了 BILBO 寄存器。在此系统中，可以使用 *LDA* 和 *LDB* 信号从 Dbus 上载入数据到寄存器 *A* 和 *B* 中。随后寄存器做加法运算，和数与进位都保存在寄存器 *C* 中。当 $B_1\&B_2$=10 时，寄存器均处于正常工作模式(*Test* = 0)，由 *LDA*, *LDB* 和 *LDC* 信号控制从总线上读入数据。为了测试加法器，我们首先令 $B_1\&B_2$=00，这时寄存器变为移位寄存器模式，并对 *A*, *B*, *C*

的初始值进行扫描。然后令 $B_1\&B_2$=01，这时寄存器 *A* 和 *B* 变为 PRPG 模式，寄存器 *C* 变为 MISR 模式。在 15 个时钟后，测试完成。然后我们令 $B_1\&B_2$=00 并扫描输出签名。

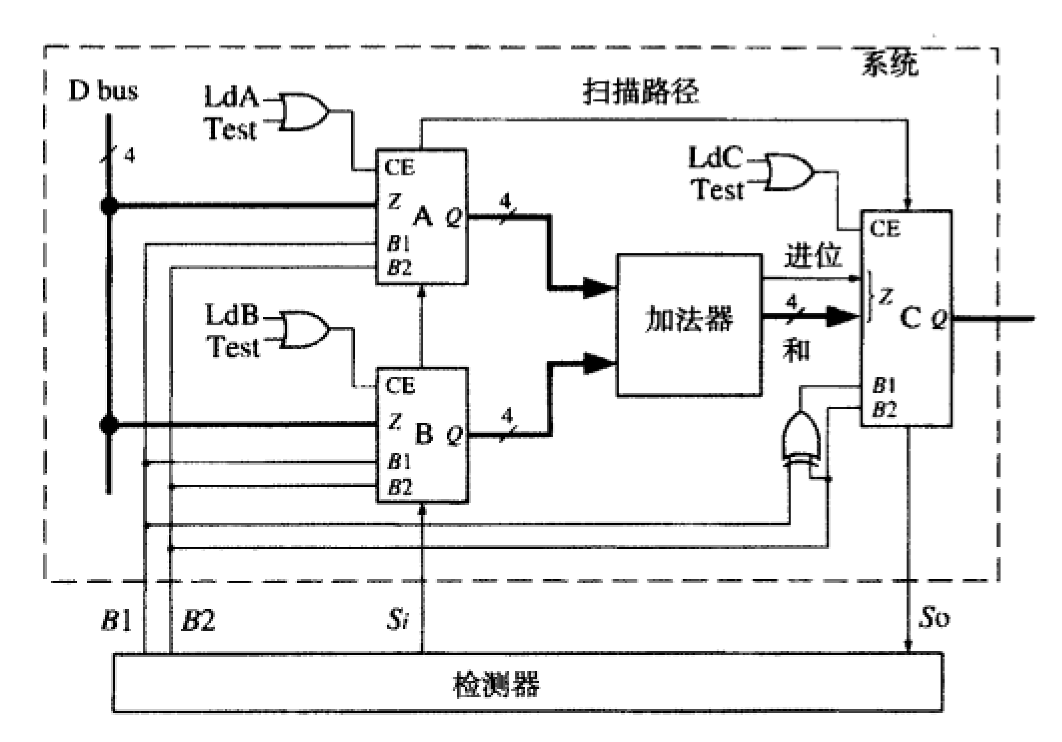


图 10.33 具有 BILBO 寄存器和测试器的系统

该系统的 VHDL 代码示于图 10.34。测试平台示于图 10.35。该系统使用了三个 BILBO 寄存器和图 8.20 所示的 4 位加法器。测试平台扫描输入一个测试矢量以初始化 BILBO 寄存器；接着在寄存器 *A* 和 *B* 用做 PRPG、寄存器 *C* 用做 MISR 时，对系统进行测试。最终签名被移出并同正确签名进行比较。

```
entity BILBO_System is
  port(Clk, LdA, LdB, LdC, B1, B2, Si: in bit;
       So: out bit;
       DBus: in bit_vector(3 downto 0);
       Output: inout bit_vector(4 downto 0));
end BILBO_System;
architecture BSys1 of BILBO_System is
  component Adder4 is
    port(A, B: in bit_vector(3 downto 0); Ci: in bit;
         S: out bit_vector(3 downto 0); Co: out bit);
  end component;
  component BILBO is
    generic(NBITS: natural range 4 to 8 := 4);
    port(Clk, CE, B1, B2, Si : in bit;
         So: out bit;
         Z: in bit_vector(1 to NBITS);
         Q: inout bit_vector(1 to NBITS));
  end component;

  signal Aout, Bout: bit_vector(3 downto 0);
  signal Cin: bit_vector(4 downto 0);
  alias Carry: bit is Cin(4);
  alias Sum: bit_vector(3 downto 0) is Cin(3 downto 0);
  signal ACE, BCE, CCE, CB1, Test, S1, S2: bit;
begin
  Test <= not B1 or B2;
  ACE <= Test or LdA;
  BCE <= Test or LdB;
  CCE <= Test or LdC;
  CB1 <= B1 xor B2;
  RegA: BILBO generic map (4) port map(Clk, ACE, B1, B2, S1, S2, DBus, Aout);
  RegB: BILBO generic map (4) port map(Clk, BCE, B1, B2, Si, S1, DBus, Bout);
  RegC: BILBO generic map (5) port map(Clk, CCE, CB1, B2, S2, So, Cin, Output);
  Adder: Adder4 port map(Aout, Bout, '0', Sum, Carry);
end BSys1;
```

图 10.34 具有 BILBO 寄存器和测试器的系统的 VHDL 代码

```
-- System with BILBO test bench

entity BILBO_test is
end BILBO_test;

architecture Btest of BILBO_test is
  component BILBO_System is
    port(Clk, LdA, LdB, LdC, B1, B2, Si: in bit;
         So: out bit;
         DBus: in bit_vector(3 downto 0);
         Output: inout bit_vector(4 downto 0));
  end component;
  signal Clk: bit := '0';
  signal LdA, LdB, LdC, B1, B2, Si, So: bit := '0';
  signal DBus: bit_vector(3 downto 0);
  signal Output: bit_vector(4 downto 0);
  signal Sig: bit_vector(4 downto 0);

  constant test_vector: bit_vector(12 downto 0) := "1000110000000";
  constant test_result: bit_vector(4 downto 0) := "01011";
begin
  clk <= not clk after 25 ns;
  Sys: BILBO_System port map(Clk,Lda,LdB,LdC,B1,B2,Si,So,DBus,Output);
  process
  begin
    B1 <= '0'; B2 <= '0';            -- Shift in test vector
    for i in test_vector'right to test_vector'left loop
      Si <= test_vector(i);
      wait until clk = '1';
    end loop;

    B1 <= '0'; B2 <= '1';            -- Use PRPG and MISR
    for i in 1 to 15 loop
      wait until clk = '1';
    end loop;

    B1 <= '0'; B2 <= '0';            -- Shift signature out
    for i in 0 to 5 loop
      Sig <= So & Sig(4 downto 1);
      wait until clk = '1';
    end loop;

    if (Sig = test_result) then      -- Compare signature
      report "System passed test.";
    else
      report "System did not pass test!";
    end if;

    wait;
  end process;
end Btest;
```

图 10.35 BILBO 系统测试平台 VHDL 代码

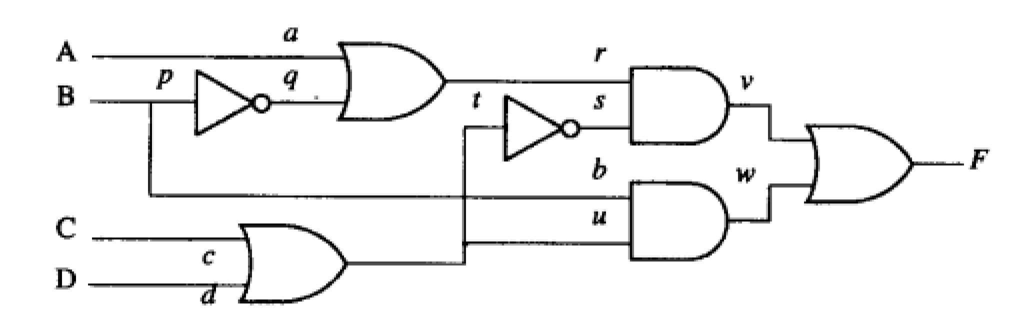
本章中，我们介绍了测试硬件的方法。涉及到了组合逻辑电路、时序逻辑电路、复杂 IC 和 PC 板。随着数字系统变得越复杂，扫描测试和内嵌自测试越显必要。在设计之初考虑设计的可测试性是非常重要的，这样才能有效、经济地进行最终硬件的测试。

习题

10.1 (a) 试确定下面待测电路的输入，以测试 u 的陷 0 故障。

(b) 对于该测试输入，试确定可测试的其他陷入故障。

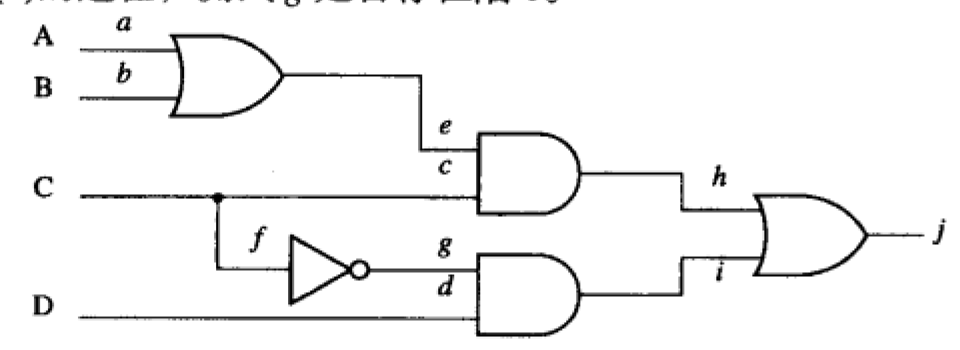
(c) 重复(a)和(b)的过程，测试 r 是否存在陷 1。



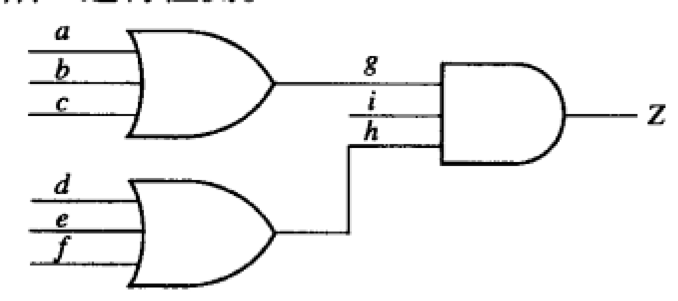
10.2 观察下面的电路。

(a) 试确定 A, B, C 和 D 的值，以测试 e 的陷 1 故障。使用该输入矢量还可以测试何种故障?

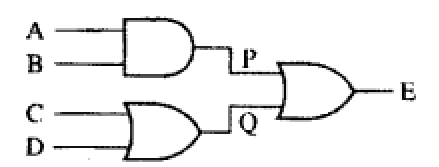
(b) 重复(a)和(b)的过程，测试 g 是否存在陷 0。



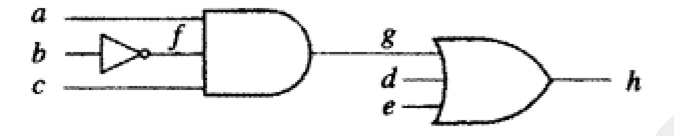
10.3 试找出最小测试集合，以测试下面电路中所有陷 1 和陷 0 故障。对于每个测试，指出是对陷 1 进行检测，还是对陷 0 进行检测。



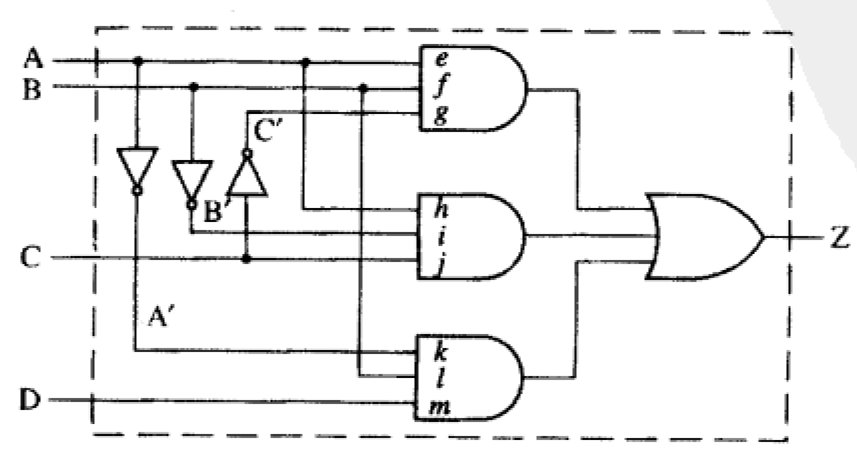
10.4 试给出最小测试向量集，以测试下面电路中所有陷入故障。列出每个测试矢量所检测的故障。



10.5 试确定 a, b, c, d 和 e 的最小测试矢量集，以测试下面电路中所有陷入故障。列出每个测试矢量所检测的故障。

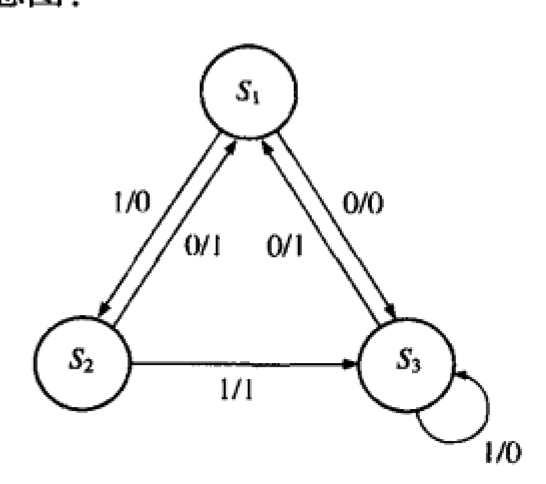


10.6 试找出最小测试向量集，以测试下面电路中所有 AND 门和 OR 门输入的陷 1 和陷 0 故障。列出每个测试矢量中 A, B, C 和 D 的值，及所检测的故障。



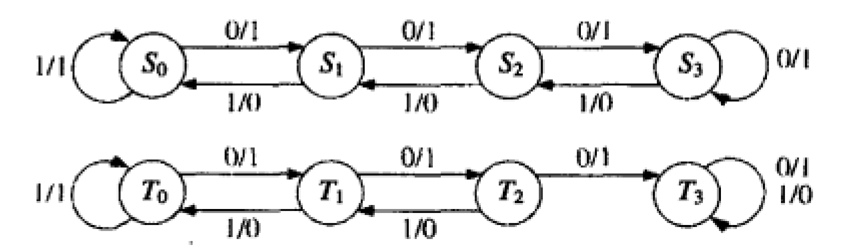
10.7 试找出测试序列，以测试图 10.7 时序电路中 b 的陷 0 故障。

10.8 一个时序电路具有如下状态图：



使用输入序列 11，并通过观察输出，可以把上图中三个状态区分开。电路具有一个复位输入 R,它可以使电路复位到状态 S_1。试给出一个可以对每个状态转移进行测试的测试序列集，并给出每个序列所测试的状态转移（在测试状态转移时，必须通过观察输出序列验证输出和下一状态的正确性）。

10.9 两个时序机的状态图如下所示。第一个图能够正确地描述时序机的功能，第二个图是对同一个时序机功能的描述，但是其描述是错误的。假设这两个时序机在各自的初始状态（S_0 和 T_0）复位，试给出可以分辨出两个时序机的最短输入序列。



10.10 在测试时序电路时，与输入输出观察法相比，扫描路径测试法的主要优点是什么？

10.11 一个扫描路径测试电路（如图 10.8 所示）有 3 个触发器、2 个输入和 2 个输出。将要测试的时序电路状态表中的一行如下所示：

		$Q_1^+Q_2^+Q_3^+$				Z_1Z_2			
$Q_1Q_2Q_3$	$X_1X_2=$	00	01	11	10	00	01	11	10
011		010	110	011	111	10	11	00	01

根据上表，完成与图 10.9 类似的时序图以说明当输入为 00, 01 和 10 时，应如何测试和验证电路的下一状态和输出。只在需要读取 Z_1Z_2 的时候对其进行输出。

10.12 (a) 使用双端口触发器，并按照图 10.8 所示格式，重新画出图 1.26 所示码转换电路。

(b) 给出能够验证图 1.24(b)状态转移表中头两行的测试序列，并使用你给出的测试序列画出时序图（与图 10.9 类似）。

10.13 (a) 写出一个双端口触发器的 VHDL 代码。

(b) 写出习题 10.12(a)答案的 VHDL 代码。

(c) 使用习题 10.12(b)中的测试序列编写测试平台，并把得到的波形同习题 10.12(b)中你的解答进行比较。

10.14 如果不使用图 10.8 中的双端口触发器，而使用标准 D 触发器（且每个 D 输入均连接一个 MUX，用以选择 D_1 或 D_2），也能完成扫描测试。请重新画出图 1.22 所示电路，用 D 触发器和 MUX 实现扫描链。测试信号(T)用以控制 MUX。

10.15 根据图 10.16，如果需要在指令寄存器中载入 011，而且在边界扫描寄存器 BSR2 中载入 1101，则起始状态为 0，终止状态为 1。试给出 *TMS* 和 *TDI* 输入对应的状态顺序。

10.16 INTEST 指令（010 码）通过把测试数据移入到边界扫描寄存器(BSR1)来测试核心逻辑，然后使用测试数据更新 BSR2。在输入单元，用此数据代替输入管脚数据。核心逻辑的输出数据在 BSR1 中被捕捉，然后移出。在此习题中，假设 BSR 有三个单元。

(a) 对于图 10.16，如果需要在指令寄存器中载入 010，在 BSR2 中载入 011，则给出所需输入 *TMS* 和 *TDI* 的序列。同时给出状态序列。起始状态为 0。

(b) 对于图 10.21 中的程序，在最后一个 BSRout 赋值语句、在 CaptureDR 状态中、在 UpdateDR 状态中应做何改变或添加，才能实现 INTEST 指令？

10.17 基于图 10.21 中的 VHDL 代码，设计一个两单元边界扫描寄存器。第一个单元为输入单元，第二个单元为输出单元。不用设计 TAP 控制器；假设所需的控制信号如 *shigt-DR*, *capture-DR* 和 *update-DR* 均可用。不用设计指令寄存器和指令译码器逻辑；假设下列信号可用：*EXT*（执行 EXTEST 指令）、*SPR*（执行采样/预载指令）和 *BYP*（执行旁路指令）。对 *BSR1* 使用两个触发器，对 BSR2 使用两个触发器，另外还使用了一个 BYPASS 触发器。除了上面提到的控制信号外，输入为 *Pin*1（从一个管脚）、*Core*2（从核心逻辑）、*TDI* 和 *TCK*；输出为 *Core*1（到核心逻辑）、*Pin*2（到管脚）和 *TDO*。所有的触发器都使用一个时钟输入 TCK。画出框图，显示出触发器、多路选择器等。然后给出每个 D 触发器的输入，每个 CE（时钟使能）和每个 MUX 控制信号的逻辑等式或连接等式。

10.18 模拟图 10.22 的边界扫描测试器并验证是否可以得到预计结果。同时改变程序，当 IC1 较低的输入与地短接时，重新进行模拟，并解释结果。

10.19 写出图 10.14(b)中边界扫描单元的 VHDL 代码。使用此边界扫描单元作为一个元件取代一些 BSR 行为描述方式代码，重写图 10.21 中的代码，并使用元件例化语句中构建该元件的一个复制 NCELLS。使用边界扫描测试器（参见图 10.22）测试你写出的新代码。

10.20 (a) 画出一个 $n = 5$ 的 LFSR 实现电路图，此 LFSR 可以生成最大长度序列。

(b) 添加逻辑电路，这样在状态序列中就会包含 00000。

(c) 给出实际状态序列。

10.21 (a) 画出一个 $n = 6$ 的 LFSR 实现电路图，此 LFSR 可以生成最大长度序列。

(b) 添加逻辑电路，这样在状态序列中就会包含 000000。

(c) 给出起始为 101010 的序列中 10 个元素。

10.22 (a) 写出与图 10.28 类似的 8 位 MISR 的 VHDL 代码。

(b) 为 6116 静态 RAM（参见图 8.15）设计一个自检电路（与图 10.25 类似）。写数据生成器按照下列顺序存储数据：00000000, 10000000, 11000000, …, 11111111, 01111111, 00111111,…,00000000。

(c) 写出设计的 VHDL 代码，并按照下面的要求仿真系统：无错误，一个错误，两个错误，三个错误。

10.23 在图 10.33 的系统中，*A*, *B*, *C* 均为 BILBO 寄存器。每个寄存器的 B_1 和 B_2 输入决定其 BILBO 操作模式如下：

B_1B_2=00，移位寄存器；B_1B_2=01，PRPG（模式生成器）

B_1B_2=10，普通系统模式；B_1B_2=11，MISR（签名寄存器）

在数据移入 A, B 和 C 时，总是先移入最低有效位。在测试模式下未使用 Dbus。给出下列操作的测试器输出序列（B_1, B_2 和 S_i）:

(1) 在 A 中载入 1011，B 中载入 1110，把 C 清零。

(2) 使用 A 和 B 作为模式生成器，C 作为签名寄存器，并在 4 个时钟周期内对系统进行测试。

(3) 把 C 寄存器输出移入测试器。

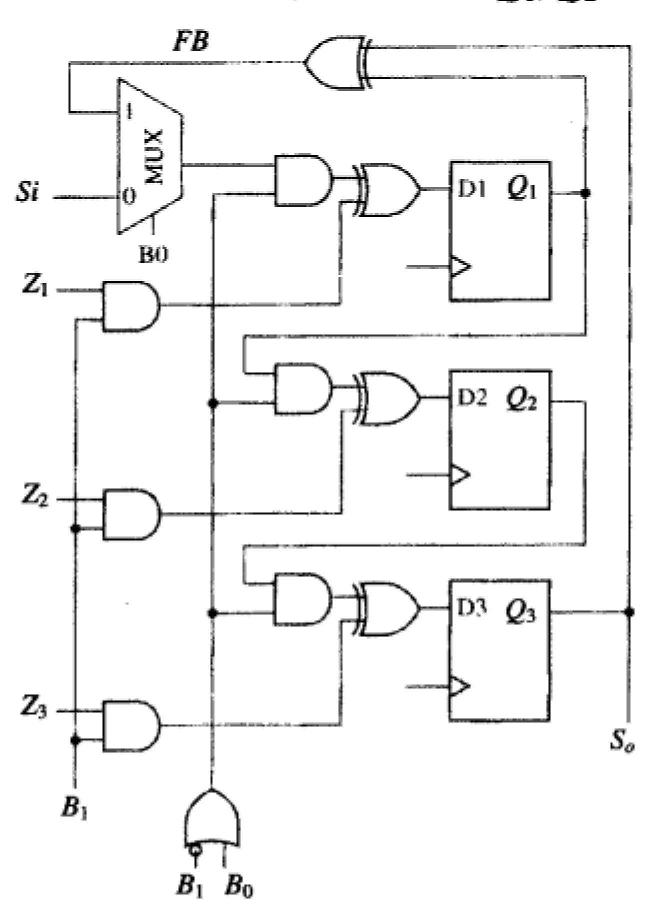
(4) 返回普通系统模式。

$$B_1B_2S_i = 000,\cdots$$

10.24 已知 BILBO 寄存器的电路如下所示。针对要求对应下面的每个模式，试给出 B_1 和 B_0 的具体值。

普通模式
移位寄存器模式
PRPG(LSFR)模式
MISR 模式

在 PRPG 模式中，假设初始状态为 001，则生成的 Q_1, Q_2 和 Q_3 的状态顺序是什么？



第 11 章　设计实例补充

在本章中，我们将进一步介绍如何使用 VHDL 和综合工具仿真和设计复杂数字系统，并列举几个实例加以说明。首先，我们将设计一个带有闹钟和秒表功能的手表；接着，对具有特定时序的内存芯片进行说明；最后，对串行数据端口收发机进行设计和阐述。

11.1　手表设计

本节中，我们将设计一块具有多功能的手表，它具有计时、闹钟和秒表功能。此手表有三个按键（B1, B2 和 B3），用来更改模式、设定时间、设定闹钟、开始和停止秒表等。按下键 B1 可以改变模式，按一下由时间变为闹钟，再按一下由闹钟变为秒表，若再被按下则恢复到时间模式。不同的模式下，按键 B2 和 B3 的功能不同，我们将在下面的内容中进行具体介绍。

11.1.1　规格说明

时间模式操作　显示时间并指出是上午还是下午(即 A.M.或 P.M.),时间显示格式为 hh:mm:ss（A 或 P）。在时间模式下，通过按下 B3 可以停止闹钟；按 B2 进行时间设定，再按则返回到时间模式。在进行时间小时和分钟设定时，每按 B3 一次就可以在小时或时钟上加 1。

闹钟模式操作　显示闹钟时间并指出是上午还是下午(即 A.M.或 P.M.),其显示格式为 hh:mm（A 或 P）。按下键 B2 进入闹钟设定状态，再按下则返回到闹钟显示界面。在进行闹钟小时和分钟设定时，每按 B3 一次就可以在小时或时钟上加 1。在闹钟显示界面下，如果按下 B3，则闹钟复位。一旦设定的闹钟时间到达，则连续响铃 50 秒，随后自动关闭，也可以在时间显示界面下按 B3 停止闹钟。

秒表模式操作　显示秒表时间，显示格式为 mm:ss.cc（其中 cc 为百万分之一秒）。按下 B2 键就开始计时，再按下 B2 键就停止计时，再按下可以接着上次继续计时，依此类推。按下 B3 键计时置零。一旦计时开始，则一直进行下去，即使手表处于时间或闹钟界面也一样。

11.1.2　设计的实现

设计框图示于图 11.1。输入模块把系统时钟降低为 100 Hz，记为 *CLK*。在输入模块中，输入键(PB1, PB2 和 PB3)是由 *CLK* 触发和同步的。每当按下 PB1, PB2 或 PB3 键时，相关信号 B1, B2 和 B3 在一个时钟周期内为 1。我们在 4.7 节中设计的单脉冲电路用于此模块中。

手表模块包含以下几个部分：主控模块、时钟模块和秒表模块。其中主控模块是控制手表的各个操作，时钟模块实现计时和闹钟功能，秒表模块实现秒表功能。100 Hz 的时钟信号 *CLK* 完成控制单元和时间寄存器的同步。控制器的状态图示于图 11.2。对应于相应的按键，状态图中生成的控制信号如下：

inch	在小时设定状态增加小时计数
incm	在分钟设定状态增加分钟计数

alarm_off	在闹钟响时，关闭闹钟
incha	增加闹钟小时计数
incma	增加闹钟分钟计数
set_alarm	闹钟开启和关闭开关
start_stop	开始或停止秒表计数器
reset	秒表计数器重启



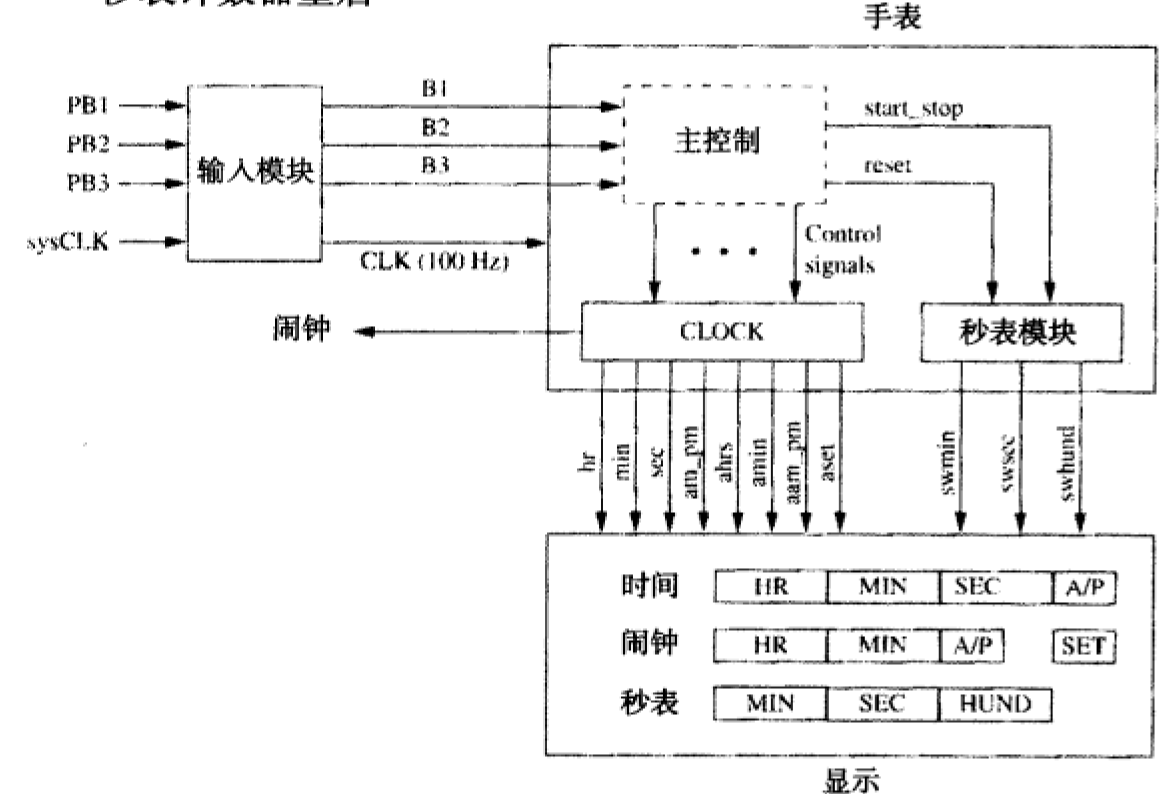


图 11.1　手表设计框图

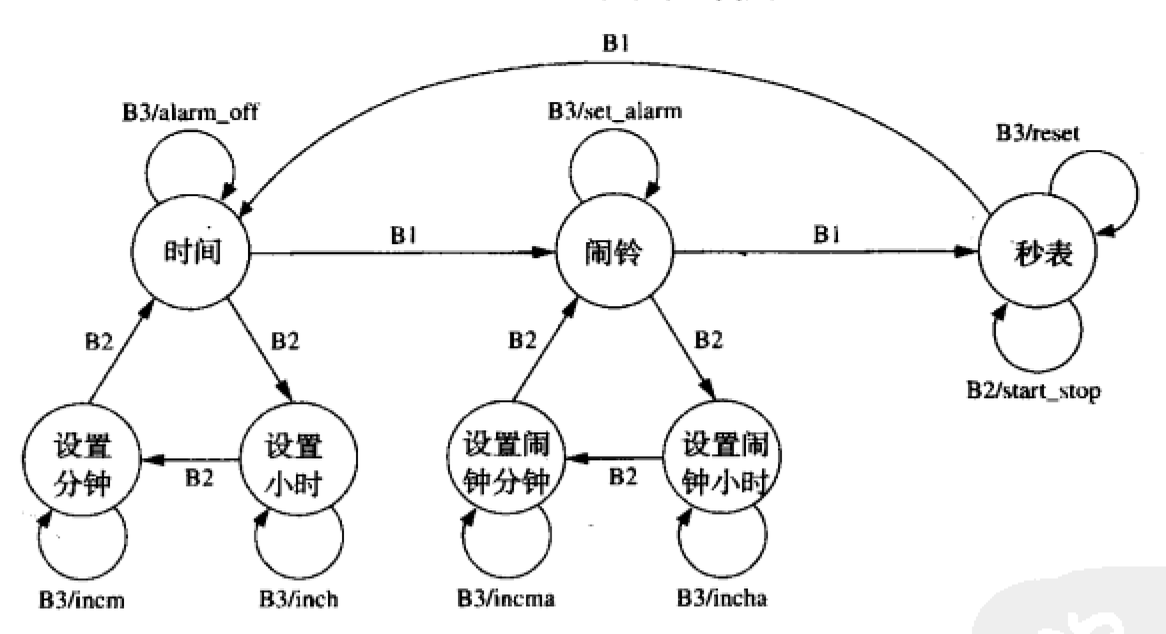


图 11.2　手表模块状态图

手表模型的 VHDL 代码示于图 11.3。该代码包括时钟模块和秒表模块，并实现了状态机。此状态机检测 B1, B2 和 B3 按键信号，并生成控制信号。VHDL 代码中所使用的信号名称如下所示：

am_pm	时间模式中 A.M.或 P.M.
aam_pm	闹钟模式中 A.M.或 P.M.
alarm_set	设置闹钟
ring	如果设置了闹钟，则表示时间计数器与设置的时间相符
hours	时间模式中小时信号
ahours	闹钟模式中小时信号

minutes	时间模式中分钟信号
aminutes	闹钟模式中分钟信号
seconds	时间模式中秒信号
swhundredths	秒表模式中百分之一秒
swseconds	秒表模式中秒信号
swminutes	秒表模式中分钟信号

```
library IEEE;
use IEEE.numeric_bit.all;

entity wristwatch is
  port(B1, B2, B3, clk: in bit;
       am_pm, aam_pm, ring, alarm_set: inout bit;
       hours, ahours, minutes, aminutes, seconds: inout unsigned(7 downto 0);
       swhundreths, swseconds, swminutes: out unsigned(7 downto 0));
end wristwatch;
architecture wristwatch1 of wristwatch is
  component clock is
    port(clk, inch, incm, incha, incma, set_alarm, alarm_off: in bit;
         hours, ahours, minutes, aminutes, seconds: inout unsigned(7 downto 0);
         am_pm, aam_pm, ring, alarm_set: inout bit);
  end component;
  component stopwatch is
    port(clk, reset, start_stop: in bit;
         swhundreths, swseconds, swminutes: out unsigned(7 downto 0));
  end component;
  type st_type is (time1, set_min, set_hours, alarm, set_alarm_hrs,
                   set_alarm_min, stop_watch);
  signal state, nextstate: st_type;
  signal inch, incm, alarm_off, set_alarm, incha, incma,
         start_stop, reset: bit;
begin
  clock1: clock port map(clk, inch, incm, incha, incma, set_alarm, alarm_off,
                         hours, ahours, minutes, aminutes, seconds, am_pm,
                         aam_pm, ring, alarm_set);
  stopwatch1: stopwatch port map(clk, reset, start_stop, swhundreths,
                                 swseconds, swminutes);
process(state, B1, B2, B3)
begin
  alarm_off <= '0'; inch <= '0'; incm <= '0'; set_alarm <= '0'; incha <= '0';
  incma <= '0'; start_stop <= '0'; reset <= '0';
  case state is
    when time1 =>
      if B1 = '1' then nextstate <= alarm;
      elsif B2 = '1' then nextstate <= set_hours;
      else nextstate <= time1;
      end if;
      if B3 = '1' then alarm_off <= '1';
      end if;
    when set_hours =>
      if B3 = '1' then inch <= '1'; nextstate <= set_hours;
      else nextstate <= set_hours;
      end if;
      if B2 = '1' then nextstate <= set_min;
      end if;
    when set_min =>
      if B3 = '1' then incm <= '1'; nextstate <= set_min;
      else nextstate <= set_min;
      end if;
      if B2 = '1' then nextstate <= time1;
      end if;
    when alarm =>
      if B1 = '1' then nextstate <= stop_watch;
      elsif B2 = '1' then nextstate <= set_alarm_hrs;
```

图 11.3　手表模块的 VHDL 代码

```
          else nextstate <= alarm;
          end if;
          if B3 = '1' then set_alarm <= '1'; nextstate <= alarm;
          end if;
        when set_alarm_hrs =>
          if B2 = '1' then nextstate <= set_alarm_min;
          else nextstate <= set_alarm_hrs;
          end if;
          if B3 = '1' then incha <= '1';
          end if;
        when set_alarm_min =>
          if B2 = '1' then nextstate <= alarm;
          else nextstate <= set_alarm_min;
          end if;
          if B3 = '1' then incma <= '1';
          end if;
        when stop_watch =>
          if B1 = '1' then nextstate <= time1;
          else nextstate <= stop_watch;
          end if;
          if B2 = '1' then start_stop <= '1';
          end if;
          if B3 = '1' then reset <= '1';
          end if;
      end case;
    end process;
    process(clk)
    begin
      if clk'event and clk = '1' then
        state <= nextstate;
      end if;
    end process;
end wristwatch1;
```

图 11.3（续） 手表模块的 VHDL 代码

时钟模块含有三个计数器。一个用于跟踪时间（*hours*, *minutes* 和 *seconds*），另一个用于存储闹钟设定的小时和分钟（*ahours* 和 *aminutes*），还有一个模 100 计数器，用于把 100 Hz 的时钟除以 100 并为秒计数器提供增 1 信号。每个计数器均使用两数字 BCD 码计数。

图 11.4 的 VHDL 代码中包含三个计数器，标记为：*sec*1，*min*1 和 *hrs*1。当在状态 99 时，模 100 计数器输出信号 *c*99，此时计数器 *sec*1 加 1。*sec*1 是计数器，并且当模 100 计数器完成一圈操作时，由于 *c*99 = 1，所以秒计数器加 1。*sec*1 是一个模 60 计数器，当到达 59 时，输出信号 *s*59。*min*1 是分钟计数器。当 *s*59 和 *c*99 同时为 1，或者在 *set_minutes* 状态中 *incm* 为 1 时，分钟计数器加 1。信号 *incmin* 用于指示何时分钟增 1，它是由按键或计数过程中的控制信号控制的。当 *min*1 到达 59 时，输出信号 *m*59。*hrs*1 为小时计数器，同时当时间从 11:59:59:99 变为 12:00:00:00 时，*hrs*1 触发触发器 *am_pm*。当 *m*59 = *s*59 = *c*99 = 1 时，或者在 *set_minutes* 状态中 *incm* 为 1 时，计数器 *hrs*1 加 1。信号 *inchr* 用于指示何时小时数增 1，它是由按键或计数过程中的控制信号控制的。

```
library IEEE;
use IEEE.numeric_bit.all;

entity clock is
  port(clk, inch, incm, incha, incma, set_alarm, alarm_off: in bit;
       hours, ahours, minutes, aminutes, seconds: inout unsigned(7 downto 0);
       am_pm, aam_pm, ring, alarm_set: inout bit);
end clock;

architecture clock1 of clock is
  component CTR_59 is
    port(clk, inc, reset: in bit; dout: out unsigned(7 downto 0);
         t59: out bit);
```

图 11.4 时钟模块的 VHDL 代码

```
    end component;
    component CTR_12 is
      port(clk, inc: in bit; dout: out unsigned(7 downto 0); am_pm: inout bit);
    end component;
    signal s59, m59, inchr, incmin, c99: bit;
    signal alarm_ring_time: integer range 0 to 50;
    signal div100: integer range 0 to 99;
    begin
      sec1: ctr_59 port map(clk, c99, '0', seconds, s59);
      min1: ctr_59 port map(clk, incmin, '0', minutes, m59);
      hrs1: ctr_12 port map(clk, inchr, hours, am_pm);
      incmin <= (s59 and c99) or incm;
      inchr <= (m59 and s59 and c99) or inch;
      alarm_min: ctr_59 port map(clk, incma, '0', aminutes, open);
      alarm_hr: ctr_12 port map(clk, incha, ahours, aam_pm);
      c99 <= '1' when div100 = 99 else '0';
      process(clk)
      begin
        if clk'event and clk = '1' then
          if c99 = '1' then div100 <= 0;     -- divide by 100 counter
          else div100 <= div100 + 1;
          end if;
          if set_alarm = '1' then
          alarm_set <= not alarm_set;
        end if;
        if ((minutes = aminutes) and (hours = ahours) and (am_pm = aam_pm)) and
            seconds = 0 and alarm_set = '1' then
          ring <= '1';
        end if;
        if ring = '1' and c99 = '1' then
          alarm_ring_time <= alarm_ring_time + 1;
        end if;
        if alarm_ring_time = 50 or alarm_off = '1' then
          ring <= '0'; alarm_ring_time <= 0;
        end if;
      end if;
    end process;
  end clock1;
```

图 11.4（续） 时钟模块的 VHDL 代码

时钟模块的 VHDL 代码还可以实现闹钟的功能，它使用计数器设定闹钟的分钟和小时。当 *alarm_set* 为 1 时，触发器 *alarm_set* 被触发。当时间计数器与闹钟设置的时间相符时，触发器 *ring* 置为 1。计数器 *Alarm_ring_time* 记录闹钟持续的秒数，且在 50 秒后，或收到 *alarm_off* 信号时，计数器清 0。

实现秒表功能的 VHDL 代码示于图 11.5。该代码包含三个计数器：百分之一秒计数器、秒计数器和分计数器。当收到 *start_stop* 信号时，计数触发器被触发。*Ctr*2 是一个模 100 BCD 计数器，当计数开始时，此计数器在每个时钟到来时加 1。在状态 99 时，*Ctr*2 计数器生成信号 *swc*99。模 100 BCD 计数器的 VHDL 代码详见图 11.6。*Sec*2 是秒计数器。当 *swc*99 = 1 时，此计数器加 1，并且在状态 59 时，此计数器生成信号 *s*59。*Min*2 是分计数器。当 *swc*99 = 1 且 *s*59 = 1 时，此计数器加 1。

```
library IEEE;
use IEEE.numeric_bit.all;

entity stopwatch is
  port(clk, reset, start_stop: in bit;
       swhundreths, swseconds, swminutes: out unsigned(7 downto 0));
end stopwatch;

architecture stopwatch1 of stopwatch is
  component CTR_59 is
```

图 11.5 秒表模块的 VHDL 代码

```
    port(clk, inc, reset: in bit; dout: out unsigned(7 downto 0); t59: out bit);
  end component;
  component CTR_99 is
    port(clk, inc, reset: in bit; dout: out unsigned(7 downto 0); t59: out bit);
  end component;
  signal swc99, s59, counting, swincmin: bit;
begin
  ctr2: ctr_99 port map(clk, counting, reset, swhundreths, swc99);
    --counts hundreths of seconds
  sec2: ctr_59 port map(clk, swc99, reset, swseconds, s59);
    --counts seconds
  min2: ctr_59 port map(clk, swincmin, reset, swminutes, open);
    --counts minutes
  swincmin <= s59 and swc99;
  process(clk)
  begin
    if clk'event and clk = '1' then
      if start_stop = '1' then
        counting <= not counting;
      end if;
    end if;
  end process;
end stopwatch1;
```

图 11.5（续）　秒表模块的 VHDL 代码

```
library IEEE;
use IEEE.numeric_bit.all;
--divide by 100 BCD counter
entity CTR_99 is
  port(clk, inc, reset: in bit; dout: out unsigned(7 downto 0); t59: out bit);
end CTR_99;

architecture count99 of CTR_99 is
  signal dig1, dig0: unsigned(3 downto 0);
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      if reset = '1' then dig0 <= "0000"; dig1 <= "0000";
      else
        if inc = '1' then
          if dig0 = 9 then dig0 <= "0000";
            if dig1 = 9 then dig1 <= "0000";
            else dig1 <= dig1 + 1;
            end if;
          else dig0 <= dig0 + 1;
          end if;
        end if;
      end if;
    end if;
  end process;
  t59 <= '1' when (dig1 = 9 and dig0 = 9) else '0';
  dout <= dig1 & dig0;
end count99;
```

图 11.6　模 100 计数器模块的 VHDL 代码

由此，我们可以直接得到模 60 计数器的 VHDL 代码（参见图 11.7）。当计数器状态为 59 时，计数器复位。

```
library IEEE;
use IEEE.numeric_bit.all;
--this counter counts seconds or minutes 0 to 59
entity CTR_59 is
  port(clk, inc, reset: in bit; dout: out unsigned(7 downto 0); t59: out bit);
end CTR_59;
```

图 11.7　模 60 计数器模块的 VHDL 代码

```
architecture count59 of CTR_59 is
  signal dig1, dig0: unsigned(3 downto 0);
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      if reset = '1' then dig0 <= "0000"; dig1 <= "0000";
      else
        if inc = '1' then
          if dig0 = 9 then dig0 <= "0000";
             if dig1 = 5 then dig1 <= "0000";
             else dig1 <= dig1 + 1;
             end if;
          else dig0 <= dig0 + 1;
          end if;
        end if;
      end if;
    end if;
  end process;
  t59 <= '1' when (dig1 = 5 and dig0 = 9) else '0';
  dout <= dig1 & dig0;
end count59;
```

图 11.7（续） 模 60 计数器模块的 VHDL 代码

当小时计数器（参见图 11.8）为 12 时，在下一时刻加 1 信号到来时，计数器变为 1。当计数器由 11 变为 12 时，*am_pm* 信号被触发。

```
library IEEE;
use IEEE.numeric_bit.all;
--this counter counts hours 1 to 12 and toggles am_pm
entity CTR_12 is
  port(clk, inc: in bit; dout: out unsigned(7 downto 0); am_pm: inout bit);
end CTR_12;

architecture count12 of CTR_12 is
  signal dig0: unsigned(3 downto 0);
  signal dig1: bit;
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      if inc = '1' then
        if dig1 = '1' and dig0 = 2 then
          dig1 <= '0'; dig0 <= "0001";
        else
          if dig0 = 9 then dig0 <= "0000"; dig1 <= '1';
          else dig0 <= dig0 + 1;
          end if;
          if dig1 = '1' and dig0 = 1 then am_pm <= not am_pm;
          end if;
        end if;
      end if;
    end if;
  end process;
  dout <= "000" & dig1 & dig0;
end count12;
```

图 11.8 小时计数器模块的 VHDL 代码

11.1.3 手表模块的测试

下面我们为手表模块编写一个测试平台（参见图 11.9）。测试平台必须包含一系列按键，100 Hz 的时钟，能够显示时间，能设置闹钟，能显示秒表计数。实际上，测试平台在整个电路中可以取代输入和显示模块。为了简化测试平台的 VHDL 代码，我们使用两个过程语句：过程 `wait1(N1)`

和过程 push(button,N)。若过程 wait1(N1)被调用，则程序等待 *N*1 个时钟。过程 push (button,N)模拟按键 *N* 次。因此 push(B2,23)模拟按 *B*2 键 23 次。push 过程对输入模块的输出进行了模拟。因此每个按键信号都持续一个时钟，并且由 *CLK* 同步。每个键被按下后，过程等待 1.2 秒。对于更长或更短的按键等待时间，我们也要进行测试。由于我们使用无符号数和 numeric_bit 包集合，所以所有的寄存器在测试电路开始运行时均为 0。如果我们使用 numeric_std 包集合，那么需要在进行仿真前把所有寄存器清 0。

测试流程如下：

1. 把时间设置到 11:58 P.M.。
2. 把闹钟设置到 12:00 A.M.。
3. 从闹钟模式切换到时间模式，等待直到时间到达午夜。
4. 在闹铃响 5 秒后，关闭闹铃。
5. 转换为秒表模式并开始计时。
6. 转换为时间模式并等待 10 秒（秒表始终在计时）。
7. 回到秒表模式并等待计时到达 1 分 2 秒。
8. 停止秒表计时，清零，并回到时间模式。

```
library IEEE;
use IEEE.numeric_bit.all;

entity testww is    -- test bench for wristwatch
  port(hours, ahours, minutes, aminutes, seconds,
       swhundreths, swseconds, swminutes: inout unsigned(7 downto 0);
       am_pm, aam_pm, ring, alarm_set: inout bit);
end testww;

architecture testww1 of testww is
  component wristwatch is
    port(B1, B2, B3, clk: in bit;
         am_pm, aam_pm, ring, alarm_set: inout bit;
         hours, ahours, minutes, aminutes, seconds: inout unsigned(7 downto 0);
         swhundreths, swseconds, swminutes: out unsigned(7 downto 0));
  end component;
  signal B1, B2, B3, clk: bit;
begin
  wristwatch1: wristwatch port map(B1, B2, B3, clk, am_pm, aam_pm, ring,
                                   alarm_set, hours, ahours, minutes, aminutes,
                                   seconds, swhundreths, swseconds, swminutes);
  clk <= not clk after 5 ms;    -- generate 100 hz clock
  process
  procedure wait1    -- waits for N1 clocks
    (N1: in integer)
    variable count: integer;
  begin
    count := N1;
    while count /= 0 loop
      wait until clk'event and clk = '1';
      count := count - 1;
      wait until clk'event and clk = '0';
    end loop;
  end procedure wait1;
  procedure push    -- simulates pushing a button N times
    (signal button: out bit; N: in integer) is
  begin
    for i in 1 to N loop
      button <= '1';
      wait1(1);
      button <= '0';
      wait1(120);    -- wait 1200 ms between pushes
```

图 11.9　手表模块测试平台

```
      end loop;
    end procedure push;
    begin
      wait1(10);    -- set time to 11:58 pm
      push(b2, 1); push(b3, 23); push(b2, 1); push(b3, 57); push(b2, 1);
      report "time should be 11:58 P.M.";
      push(b1, 1);    -- set alarm to 12:00 am
      push(b2, 1); push(b3, 24); push(b2, 2); push(b3, 1); push(b1, 2);
      report "alarm should be set to 12:00 A.M.";
      wait until hours = "00010010" and seconds = "00000101";
      push(b3, 1);    -- turn alarm off at 12 hours and 5 seconds
      push(b1, 2);    -- run stopwatch, go to time mode, go back to stopwatch
      push(b2, 1); wait1(120); push(b1, 1); wait1(1000); push(b1, 2);
      wait until swminutes = "00000001" and swseconds = "00000010";
        --stop stopwatch after 1 min. and 2 sec., then reset
      report "stopwatch should read 1 min. 2 sec.";
      push(b2, 1); push(b3, 1); push(b1, 1);
      wait;
    end process;
  end testwwl;
```

图 11.9（续） 手表模块测试平台

我们使用以下命令对上面所示测试流程进行仿真。

```
vsim -t 1ms testww    -- set simulator resolution to 1 ms
add list -hex hours minutes seconds am_pm
              ahours aminutes aam_pm ring
add list b1 b2 b3 wristwatch1/state
add list -hex swminutes swseconds -notrigger swhundredths
run 300000 ms
```

测试结果显示手表模块可以完成预设功能。当使用 Xilinx Spartan 3 FPGA 实现手表模块时，需要 87 个片、80 个寄存器和 158 个 4 输入 LUT。对于完整的手表模块设计，我们还需要编写输入和显示模块程序。

11.2 存储器时序模型

当我们设计一个由多个元件组成的复杂数字系统时，为了使系统的各个组成部分能够更好地协调工作，该系统必须要满足很多时序约束。例如，如果我们要存储器与微处理器的总线接口，则必须满足所有总线接口的时序规范。为了使用 VHDL 对这一系统进行仿真，我们必须对每个组成元件建立准确的时序模型。本节中我们将对一个静态 RAM 存储器建立时序模型。我们说明从芯片厂商的规范开始，到考虑时序参数的 VHDL 模型的建立过程。这种时序模型对于芯片系统(SoC)设计是很有好处的。

图 11.10 给出了 6116 静态 RAM 的框图。它可以存储 2 k 字节的数据。该存储器有 16 384 个单元，排列成 128×128 存储矩阵。该 RAM 含有地址译码器和一个内存阵列。地址译码分为行译码器和列译码器。11 条地址线用于对 2^{11} 字节的数据寻址，分为两组，一组用于行译码，另一组用于列译码。地址线 A0 ~ A3 可以一次选择存储矩阵中的 8 列，这是因为每个地址都有 8 条数据线。A4 ~ A10 用来选择矩阵 128 行中的一行。数据输出在连接到 I/O 线之前要通过三态缓冲器，这些缓冲器只有在进行存储器读操作时才起作用。

片选信号($\overline{CS}$)、输出允许信号($\overline{OE}$)和写允许信号($\overline{WE}$)之间的时序关系示于如图 11.10。这些信号的功能在表 8.7 中已经加以介绍。当 $\overline{CS}$ 为高电平时（即无效），两个与门都有一个输入信号为 0，这样三态缓冲的控制电压为低电平，输出为高阻（Hi-Z）。同理，当 $\overline{OE}$ 为高电平时，即使片选有效，三态控制仍为无效，输出仍为高阻。当片选和 $\overline{WE}$ 都有效时，就开始进行写操作，I/O 线上的数据开始写入 RAM。当片选和 $\overline{OE}$ 都有效，而 $\overline{WE}$ 无效时，则进行读操作，RAM 的内容出现在 I/O 线上。

在 8.7 节中，我们介绍了一些静态 RAM 存储器模块，但该模块并没有考虑时序问题。当使用存储器芯片进行系统设计时，必须要考虑到时序图和时间参数。本节中，我们将给出具有特定时间参数的存储器芯片的仿真模型。

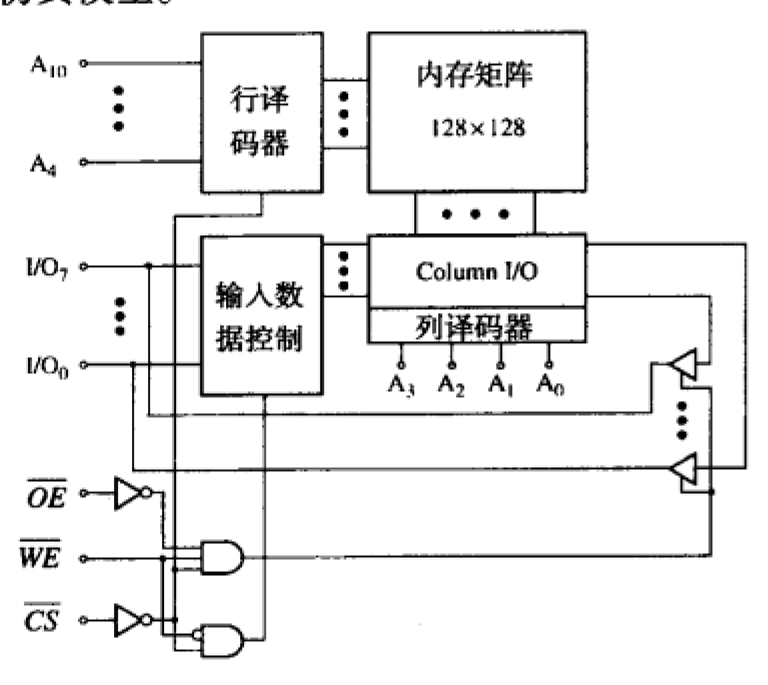


图 11.10 6116 静态 RAM 框图

让我们考虑 6116 CMOS 静态 RAM 芯片，其读写周期的时间参数定义如表 11.1 所示。表中给出了 6116 SA-15 RAM 的参数指标，其存取时间为 15 ns。表中的破折号表示该参数无关或者生产商没有提供。

表 11.1 CMOS 静态 RAM 6116 SA-15 的时序特征参数

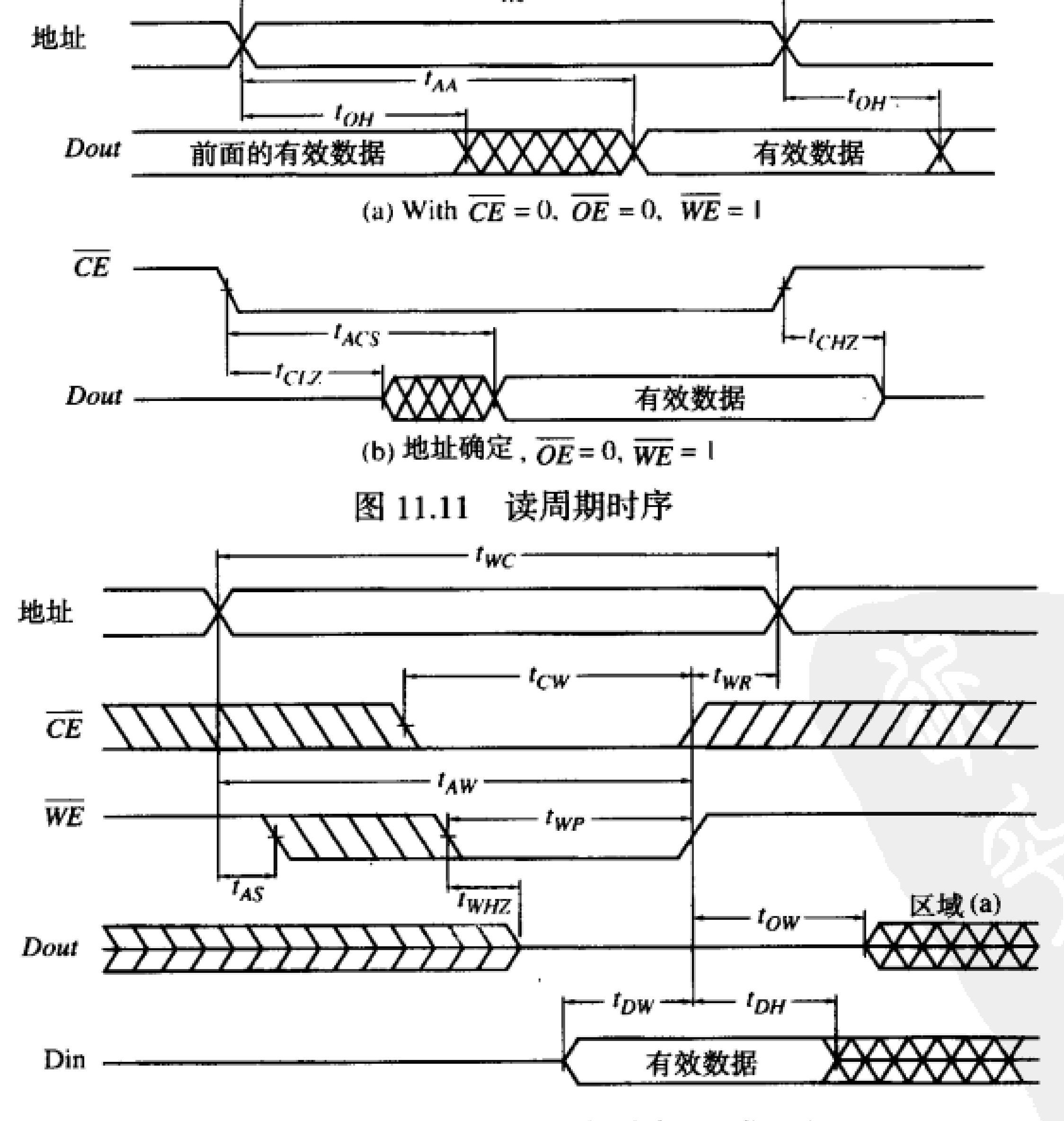
参数	符号	时序特征	
		最小	最大
读周期时间	t_{RC}	15	–
地址有效后读取时间	t_{AA}	–	15
片选有效后读取时间	t_{ACS}	–	15
片选有效到输出低阻时间	t_{CLZ}	5	–
输出允许到输出有效时间	t_{OE}	–	10
输出允许到输出低阻时间	t_{OLZ}	0	–
片选无效到输出高阻时间	t_{CHZ}	2*	10
输出不允许到输出高阻时间	t_{OHZ}	2*	8
地址改变后的保持时间	t_{OH}	5	–
写周期时间	t_{WC}	15	–
片选到写结束时间	t_{CW}	13	–
地址有效到写结束时间	t_{AW}	14	–
地址建立时间	t_{AS}	0	–
写脉冲宽度	t_{WP}	12	–
写恢复时间	t_{WR}	0	–
写允许到输出高阻时间	t_{WHZ}	–	7
写结束后数据有效时间	t_{DW}	12	–
写结束后数据保持时间	t_{DH}	0	–
写结束后输出有效时间	t_{OW}	0	–

*估计值，非生产商提供。

图 11.11(a)给出了 $\overline{CS}$ 和 $\overline{OE}$ 都为低电平时的读周期时序，此时地址还没有改变。当地址发生改变后，旧数据在存储器的输出端仍保持 t_{OH} 时间，然后有一个过渡时间（用交叉线表示）用以改变数据。地址有效后，过了 t_{AA} 读取时间，新的数据就稳定在存储器输出端。在读周期时间 t_{RC} 内，地址必须是稳定的。

图 11.11(b)给出的时序图是 $\overline{OE}$ 为低电平，且 $\overline{CS}$ 变为低电平前地址是稳定的。如果 $\overline{CS}$ 为高电平，则 *Dout* 为高阻状态，图中由'0'和'1'中间的一条线来表示。如果 $\overline{CS}$ 为低电平，则过了 t_{CLZ} 时间后 *Dout* 就离开高阻状态。此时有一段过渡时间用于改变数据。在 $\overline{CS}$ 有效后，经过 t_{ACS} 时间新数据就稳定了。在 $\overline{CS}$ 变为高电平后，经过 t_{CHZ} 时间 *Dout* 重新变为高阻。

图 11.12 给出一个由 $\overline{WE}$ 控制的写周期时序，在这一周期内 $\overline{OE}$ 一直为低电平。这里假设了 $\overline{CS}$ 是在 $\overline{WE}$ 之前（或同时）变为低电平，在 $\overline{WE}$ 之前（或同时）又变回高电平。图中 $\overline{CS}$ 上的交叉线表示在该区间 $\overline{CS}$ 可以从高电平变为低电平（或者从低电平变回高电平）。$\overline{WE}$ 变成低电平之前，在地址建立时间 t_{AS} 内，地址必须是稳定的。在时间 t_{WHZ} 之后，缓冲器的数据输出进入高阻状态，输入数据就可以放到 I/O 线上。$\overline{WE}$ 变回高电平之前，在建立时间 t_{DW} 内，要写入存储器的数据必须稳定，并且之后的保持时间 t_{DH} 内也必须保持稳定。$\overline{WE}$ 变回高电平后，在 t_{OW} 时间内，地址必须保持稳定。$\overline{WE}$ 变回高电平后，存储器就切换为读模式。在经过 t_{OW} (min)时间，*Dout* 通过区间(a)的过渡期，最终与刚存的数据相同。如果地址发生改变，或者 $\overline{CS}$ 变成高电平，则 *Dout* 可能再次改变。为了避免区间(a)中的总线冲突，*Din* 应该为高阻或者与 *Dout* 相同。



(a) With $\overline{CE}$ = 0, $\overline{OE}$ = 0, $\overline{WE}$ = 1

(b) 地址确定，$\overline{OE}$ = 0, $\overline{WE}$ = 1

图 11.11　读周期时序

图 11.12　$\overline{WE}$ 控制的写周期时序（$\overline{OE}$ =0）

图 11.13 给出了一个由 $\overline{CS}$ 控制的写周期时序，在这一周期内 $\overline{OE}$ 一直为低电平。这里假设了 $\overline{WE}$ 是在 $\overline{CS}$ 之前（或同时）变为低电平，在 $\overline{CS}$ 之前（或同时）又变回高电平。$\overline{CS}$ 变成低电平之前，在地址建立时间 t_{AS} 内，地址必须是稳定的。$\overline{CS}$ 变回高电平之前，在建立时间 t_{DW} 内，要写入存储器的数据必须保持稳定，并且之后在保持时间 t_{DH} 内也必须保持稳定。$\overline{CS}$ 变为高电平后，在 t_{WR} 时间内，地址必须保持稳定。注意到这一写周期与 $\overline{WE}$ 控制的写周期很相似。不管哪种情况，存储器的写操作都是在 $\overline{CS}$ 或 $\overline{WE}$ 为低电平时进行的，并且只要有一个变为高电平就结束该写周期。

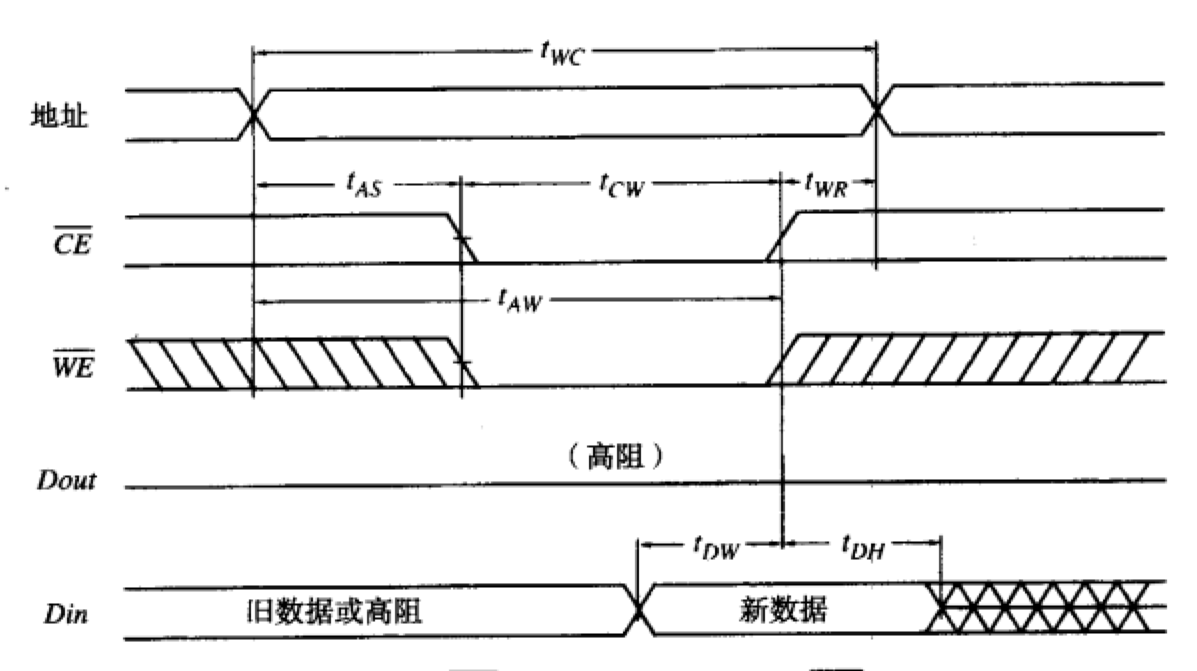


图 11.13　$\overline{CS}$ 控制的写周期时序($\overline{OE}=0$)

下面，根据图 11.11、图 11.12 和图 11.13 的读写周期时序关系，我们重新考虑图 8.15 的 RAM 模块。假设 $\overline{OE}$ =0。RAM 的 VHDL 时间模型示于图 11.14，该模型中使用了 generic 语句设置了重要时间参数的默认值。为了避免由惯性延迟引发的抵消问题，我们都用了传输延迟。RAM 进程等待 *CS_b*, *WE_b* 或地址的变化。当 *CS_b* 为'0'时 *WE_b* 的上升沿到来，或者当 *WE_b* 为'0'时 *CS_b* 的上升沿到来，表示写操作的结束，此时数据已写入 RAM，并过 t_{OW} 时间后把数据读回。如果当 *CS_b* 为'0'时 *WE_b* 的下降沿到来，那么 RAM 切换为写模式，数据输出变为高阻态。

```
-- memory model with timing (OE_b=0)
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity static_RAM is
  generic(constant tAA:  time := 15 ns;   -- 6116 static CMOS RAM
          constant tACS: time := 15 ns;
          constant tCLZ: time :=  5 ns;
          constant tCHZ: time :=  2 ns;
          constant tOH:  time :=  5 ns;
          constant tWC:  time := 15 ns;
          constant tAW:  time := 14 ns;
          constant tWP:  time := 12 ns;
          constant tWHZ: time :=  7 ns;
          constant tDW:  time := 12 ns;
          constant tDH:  time :=  0 ns;
          constant tOW:  time :=  0 ns);
```

图 11.14　6116 静态 CMOS RAM 时序仿真模块

```
  port(CS_b, WE_b, OE_b: in std_logic;
       Address: in unsigned(7 downto 0);
       IO: inout unsigned(7 downto 0) := (others => 'Z'));
end Static_RAM;

architecture SRAM of Static_RAM is
  type RAMtype is array(0 to 255) of unsigned(7 downto 0);
  signal RAM1: RAMtype := (others => (others => '0'));
begin
  RAM: process (CS_b, WE_b, Address)
  begin
    if CS_b='0' and WE_b='1' and Address'event then
    -- read when address changes
      IO <= transport "XXXXXXXX" after tOH,
            Ram1(to_integer(Address)) after tAA; end if;
    if falling_edge(CS_b)and WE_b='1' then
    -- read when CS_b goes low
      IO <= transport "XXXXXXXX" after tCLZ,
            Ram1(to_integer(Address)) after tACS; end if;
    if rising_edge(CS_b) then    -- deselect the chip
      IO <= transport "ZZZZZZZZ" after tCHZ;
      if We_b='0' then -- CS-controlled write
        Ram1(to_integer(Address'delayed)) <= IO; end if;
    end if;
    if falling_edge(WE_b) and CS_b='0' then    -- WE-controlled write
      IO <= transport "ZZZZZZZZ" after tWHZ; end if;
    if rising_edge(WE_b) and CS_b='0' then
      Ram1(to_integer(Address'delayed)) <= IO'delayed;
      IO <= transport IO'delayed after tOW;    -- read back after write
        -- IO'delayed is the value of IO just before the rising edge
    end if;
  end process RAM;

  check: process
  begin
    if NOW /= 0 ns then
      if address'event then
        assert (address'delayed'stable(tWC))    -- tRC = tWC assumed
          report "Address cycle time too short"
          severity WARNING;
      end if;
    -- The following code only checks for a WE_b controlled write:
      if rising_edge(WE_b) and CS_b'delayed = '0' then
        assert (address'delayed'stable(tAW))
          report "Address not valid long enough to end of write"
          severity WARNING;
        assert (WE_b'delayed'stable(tWP))
          report "Write pulse too short"
          severity WARNING;
        assert (IO'delayed'stable(tDW))
          report "IO setup time too short"
          severity WARNING;
        wait for tDH;
          assert (IO'last_event >= tDH)
          report "IO hold time too short"
          severity WARNING;
      end if;
    end if;
    wait on CS_b, WE_b, Address;
  end process check;
end SRAM;
```

图 11.14（续） 6116 静态 CMOS RAM 时序仿真模块

若出现 *CS_b* 上升沿，则 RAM 不被选中，数据输出在指定的延迟后进入高阻状态。相反，若出现 *CS_b* 下降沿，*WE_b* 为'1'，则 RAM 处于读模式，t_{CLZ}(min)时间后，数据总线可脱离高阻状态，但在 t_{ACS}(max)之前不能保证有效的数据输出。在这两个时间之间的区域是过渡区。由于在过

渡区总线状态是未知的，所以 I/O 线上的输出为'X'。如果地址改变时 RAM 处于读模式（图 11.11(a)），则旧数据保持一段时间 t_{OH}，随后进入到过渡区，直到时间 t_{AA} 之后才可以从存储器中读出有效的新数据。

Check 进程与 RAM 进程是并发执行的，用来检测存储器的一些时序关系是否正确。*NOW* 是一个提前定义的变量，它等于当前时间（VHDL 之所以提供 *NOW* 是为了读取当前仿真时间。它是一个提前定义的函数。在仿真过程中，在不同时间调用会返回不同的值）。为了避免误报错误信息，如果 *NOW* = 0 或者芯片未被选中，则检查不会进行。当地址发生改变时，check 进程将检查地址在写周期时间内(t_{WC})是否为稳定的。如果地址不稳定，则发出一个警告信息。如果在进行检测时，地址恰好发生改变，*Address'stable* (t_{WC})将总是返回 FALSE。因此，必须用 *Adderess'delayed* 代替 *Address*。这样 *Address* 将被延迟 Δ 时间，在 *Address* 改变前，完成地址检测稳定。下面我们对写周期的时序特性进行检查。首先，我们验证地址在 t_{AW} 内是否稳定，然后检查 *WE_b* 在 t_{WP} 内是否为低电平。最后，检查数据的建立和保持时间。

RAM 时序模型的部分测试 VHDL 代码示于图 11.15。该代码运行一个写周期后，再运行两个读周期。在各个周期之间，RAM 不被选中。测试结果示于图 11.16。对于输入同时发生改变和违反时序关系的情况，我们也进行了测试，但这些测试结果没有在这里列出。

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity RAM_timing_tester is
end RAM_timing_tester;

architecture test1 of RAM_timing_tester is
  component static_RAM is
    port(CS_b, WE_b, OE_b: in std_logic;
         Address: in unsigned(7 downto 0);
         IO: inout unsigned(7 downto 0));
  end component Static_RAM;
  signal Cs_b, We_b: std_logic := '1';    -- active low signals
  signal Data: unsigned(7 downto 0)  := "ZZZZZZZZ";
  signal Address: unsigned(7 downto 0):= "00000000";
begin
  SRAM1: Static_RAM port map(Cs_b, We_b, '0', Address, Data);
  process
  begin
    wait for 20 ns;
    Address <= "00001000";                -- WE-controlled write
    Cs_b <= transport '0', '1' after 50 ns;
    We_b <= transport '0' after 8 ns, '1' after 40 ns;
    Data <= transport "11100011" after 25 ns, "ZZZZZZZZ" after 55 ns;

    wait for 60 ns;
    Address <= "00011000";               -- RAM deselected
    wait for 40 ns;
    Address <= "00001000";               -- Read cycles
    Cs_b <= '0';
    wait for 40 ns;
    Address <= "00010000";
    Cs_b <= '1' after 40 ns;
    wait for 40 ns;
    Address <= "00011000";               -- RAM deselected
    wait for 40 ns;
    report "DONE";
  end process;
end test1;
```

图 11.15　RAM 时间模型的测试 VHDL 代码

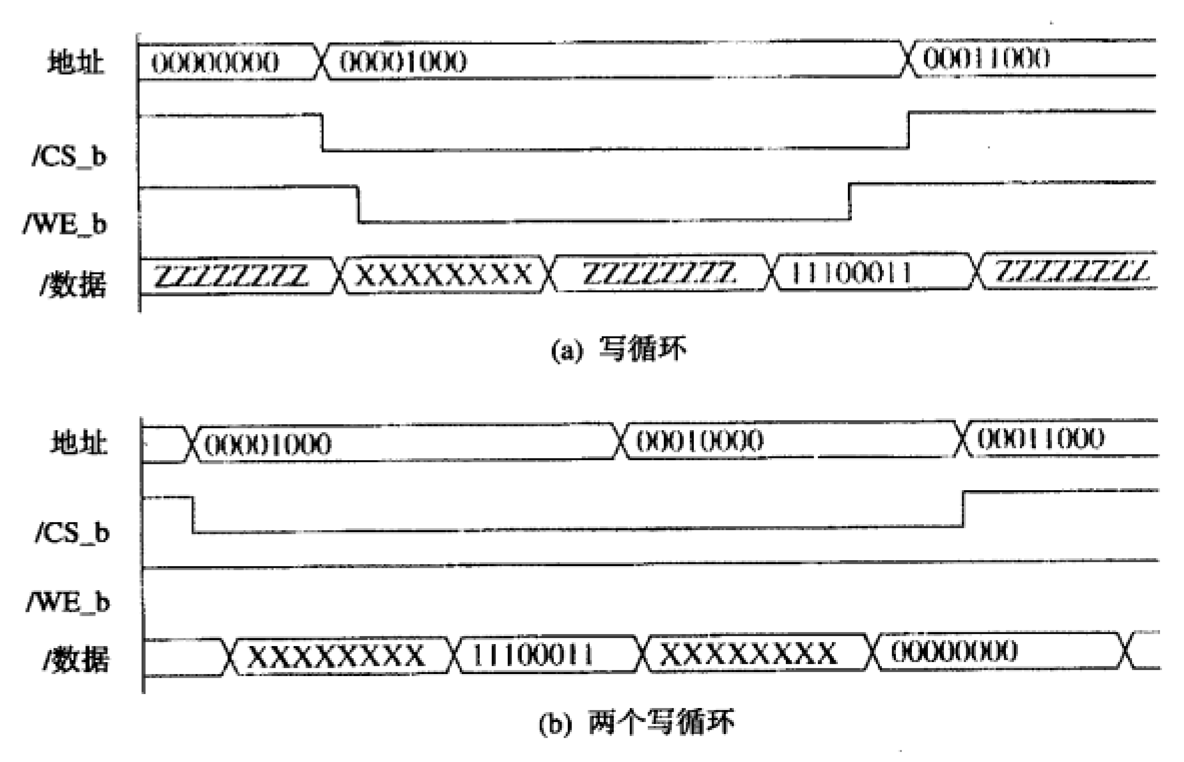


图 11.16 RAM 时间模型测试结果

11.3 通用异步收发机

大多数计算机和微控制器都有一个或多个串行数据端口，与串行输入/输出设备进行通信，例如键盘和串行打印机。通过在串行端口上使用调制解调器，串行数据可以通过电话线进行长距离的收发（参见图 11.17）。用于收发串行数据的串行通信接口通常称为 UART（通用异步收发机）。在图 11.17 中，*RxD* 为接收到的串行数据信号，*TxD* 为发送的串行数据信号。

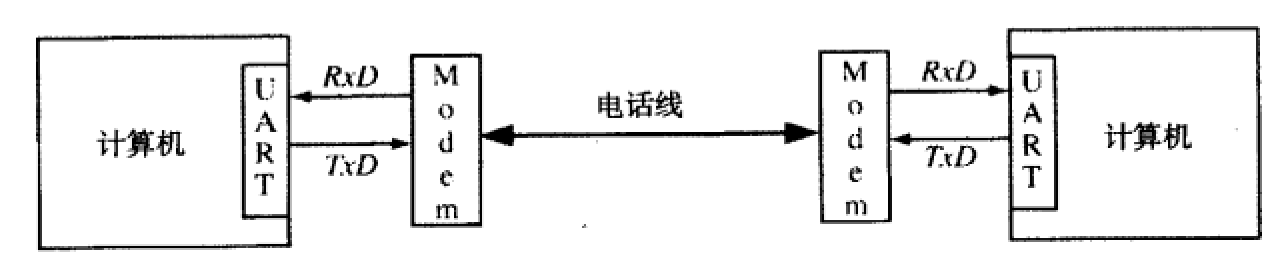


图 11.17 串行数据传输

串行数据的标准格式如图 11.18 所示。由于没有时钟线，所以数据(*D*)是异步传输的，每次传输一个字节。当没有数据传输时，*D* 保持为高电平。为了标识传输的开始，*D* 在刚开始的一个比特时间内为低电平，这个比特位称为起始位。紧接着传输 8 比特的数据，从最低有效位开始传输。传输文本常用 ASCII 码。ASCII 码的每个字符都用 7 位码表示，第八位可以用做奇偶校验位。例如，字母 U 的代码为 1010101，第八位奇偶校验位为 0，所以代码中有偶数个 1。在 8 比特都传输完毕后，*D* 必须重新变为高电平，并持续至少 1 比特的时间，该比特位称为停止位。然后，我们可以在之后的任何时间开始传输其他字符。

每秒传输的比特数称为**波特率**（baud rate）。

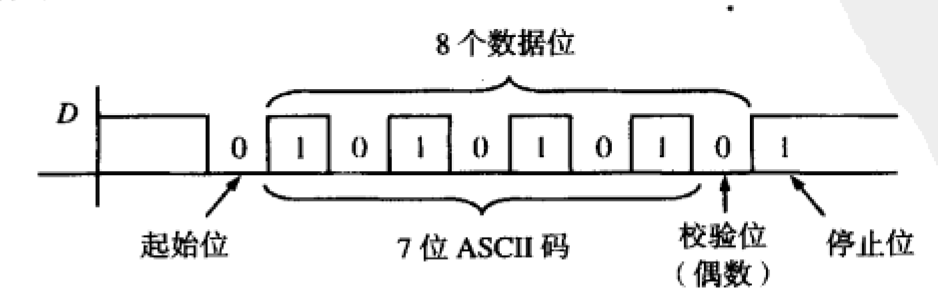


图 11.18 串行数据的标准格式

在发送时，UART 把 8 位并行数据转换为串行数据流，此数据流由 1 个起始位（逻辑 0）、8 个数据位（最低有效位在前）、一个或多个停止位（逻辑 1）组成。在接收时，UART 对起始位进行检测，接收 8 个数据位并把数据转换为并行格式，然后检测停止位。由于未发送时钟信号，所以 UART 必须用当地时钟对接收到的数据流进行同步。

下面我们设计一个简单的 UART，它与微控制器 MC6805、MC6811 和其他微控制器中使用的 UART 类似。图 11.19 给出 UART 如何与 8 位数据总线相连。图中使用了 6 个 8 位寄存器，定义如下：

RSR	接收移位寄存器
RDR	接收数据寄存器
TDR	发送数据寄存器
TSR	发送移位寄存器
SCCR	串行通信控制寄存器
SCSR	串行通信状态寄存器

假设 UART 连接在微控制器的数据和地址总线上，所以 CPU 可以对寄存器进行读写，寄存器 *RDR*, *TDR*, *SCCR* 和 *SCSR* 是内存映射的，每个寄存器都赋有一个地址在微控制器的内存空间。*RDR*, *SCSR* 和 *SCCR* 通过三态缓冲器驱动数据总线，*TDR* 和 *SCCR* 从数据总线上可以载入数据。

除了寄存器以外，UART 还有三个重要组成部分：波特率生成器、接收机控制器和发送机控制器。波特率生成器可以对系统时钟进行分频，提供周期为 1 比特时间的位时钟（*BClk*）和频率为 *BClk* 的 8 倍频的 *BclkX8* 时钟。

当 *TDR* 为空时，要对 *SCSR* 中 *TDRE* 标志位（发送数据寄存器为空）置 1。若微控制器准备好发送数据时，则

1. 微控制器等待 *TDRE* = '1'，随后 *TDR* 中载入一个字节数据，并把 *TDRE* 清零。
2. UART 把数据从 *TDR* 发送到 *TSR*，并置位 *TDRE*。
3. UART 输出一比特时间的开始位（'0'），然后通过对 *TSR* 右移一位一位地发送出 8 位数据比特，最后发送停止位（'1'）。

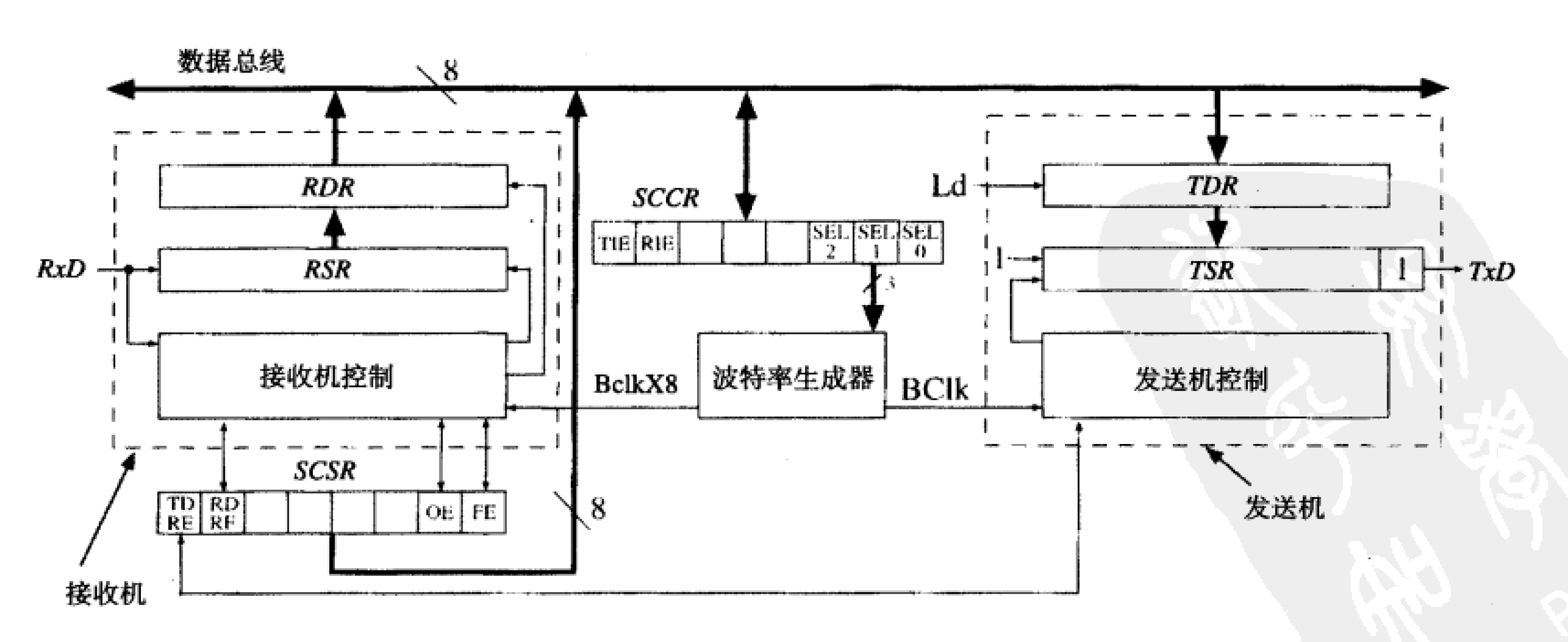


图 11.19　UART 模块框图

图 11.20 给出了发射机的 SM 图。这一 SM 图的时序机是由微控器的系统时钟(*CLK*)驱动的。在 IDLE 状态, SM 一直等到 *TDR* 被载入数据且 *TDRE* 清零。在 SYNCH 状态，直到位时钟上升沿(*Bclk*↑)到来，并把 *TSR* 的最低位清零，一个比特时间内发送'0'。在 TDATA 状态，在每个 *Bclk*↑时，TSR 都进行右移，发送下一个数据，并且位计数器(*Bct*)加 1。当 *Bct* = 9 时，8 个数据位和一个停止位均发送完毕。*Bct* 清零，SM 返回状态 IDLE。

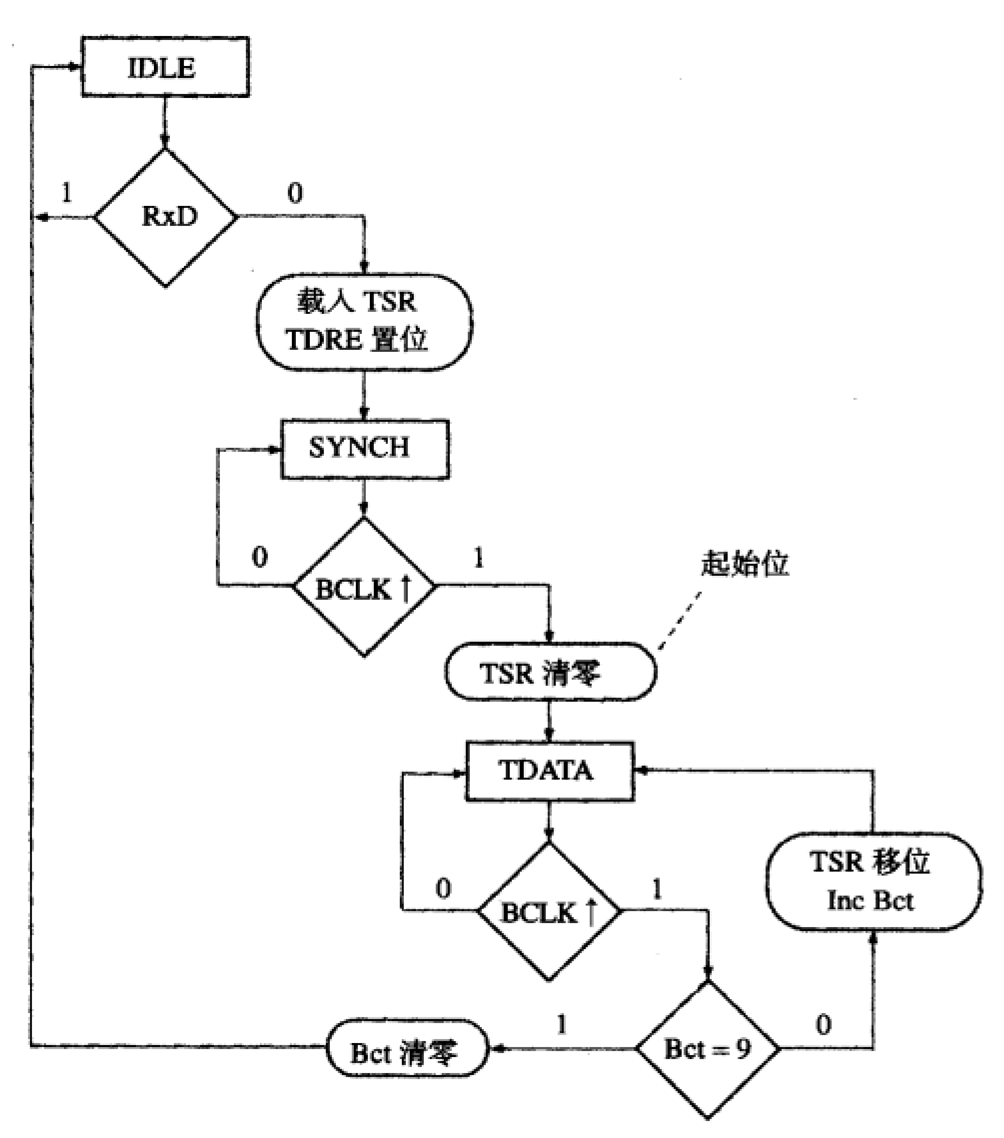


图 11.20 UART 发送机 SM 图

UART 发送机的 VHDL 代码（参见图 11.21）是基于图 11.20 的 SM 图的。如果使用了 `std_logic_vector` 数据类型，则没有必要使用语句 use IEEE.numeric_std.all。在这里和另一个 UART 模块里，我们都使用了无符号数据类型。发送机是由 *TDR*, *TSR* 寄存器和发送机控制器构成的，与 *TDRE* 和数据总线(DBUS)接口。第一个进程表示一个生成下一状态信号和控制信号的组合逻辑网络。第二个进程在时钟上升沿对寄存器进行更新。在时钟 *Bclk* 上升沿到来时，信号 *Bclk_rising* 变为'1'，并持续一个系统时钟周期。为了生成 *Bclk_rising*，我们把 *Bclk* 存储在 *Bclk_Dlayed* 寄存器中。当 *Bclk* 的当前值为'1'，而前一时刻的值为'0'时，把 *Bclk_rising* 置 1，即

```
Bclk_rising <= Bclk and not Bclk_Dlayed;
```

UART 接收机的工作流程如下：

1. 当 UART 检测到起始位时，UART 继续读取其他位并通过移位把它们移入到 *RSR* 中。
2. 当所有的数据位和停止位都接收完毕后，*RSR* 中数据载入到 *RDR* 中，*SCSR* 中 *RDRF*（*RDR* 寄存器满）标志位置 1。
3. 微控制器检测 *RDRF* 标志位，如果为 1，则读取 *RDR* 中数据并把 *RDRF* 标志位清零。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all; -- use this if unsigned type is used.

entity UART_Transmitter is
  port(Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
       DBUS: in unsigned(7 downto 0);
       setTDRE, TxD: out std_logic);
end UART_Transmitter;

architecture xmit of UART_Transmitter is
  type stateType is (IDLE, SYNCH, TDATA);
  signal state, nextstate: stateType;
  signal TSR: unsigned(8 downto 0);    -- Transmit Shift Register
  signal TDR: unsigned(7 downto 0);    -- Transmit Data Register
  signal Bct: integer range 0 to 9;    -- counts number of bits sent
  signal inc, clr, loadTSR, shftTSR, start: std_logic;
  signal Bclk_rising, Bclk_Dlayed: std_logic;
begin
  TxD <= TSR(0);
  setTDRE <= loadTSR;
  Bclk_rising <= Bclk and (not Bclk_Dlayed);
  -- indicates the rising edge of bit clock

  Xmit_Control: process(state, TDRE, Bct, Bclk_rising)
  begin
    inc <= '0'; clr <= '0'; loadTSR <= '0'; shftTSR <= '0'; start <= '0';
    -- reset control signals
    case state is
      when IDLE =>
        if (TDRE = '0') then
          loadTSR <= '1'; nextstate <= SYNCH;
        else nextstate <= IDLE;
        end if;
      when SYNCH =>    -- synchronize with the bit clock
        if (Bclk_rising = '1') then
          start <= '1'; nextstate <= TDATA;
        else nextstate <= SYNCH;
        end if;
      when TDATA =>
        if (Bclk_rising = '0') then nextstate <= TDATA;
        elsif (Bct /= 9) then
          shftTSR <= '1'; inc <= '1'; nextstate <= TDATA;
        else clr <= '1'; nextstate <= IDLE;
        end if;
    end case;
  end process;

  Xmit_update: process(sysclk, rst_b)
  begin
    if (rst_b = '0') then
      TSR <= "111111111"; state <= IDLE; Bct <= 0; Bclk_Dlayed <= '0';
    elsif (sysclk'event and sysclk = '1') then
      state <= nextstate;
      if (clr = '1') then Bct <= 0;
      elsif (inc = '1') then
        Bct <= Bct + 1;
      end if;
      if (loadTDR = '1') then TDR <= DBUS;
      end if;
      if (loadTSR = '1') then TSR <= TDR & '1';
      end if;
      if (start = '1') then TSR(0) <= '0';
      end if;
      if (shftTSR = '1') then TSR <= '1' & TSR(8 downto 1);
      end if;
```

图 11.21　UART 发送机 VHDL 代码

```
      -- shift out one bit
      Bclk_Dlayed <= Bclk;    -- Bclk delayed by 1 sysclk
    end if;
  end process;
end xmit;
```

图 11.21（续） UART 发送机 VHDL 代码

从 *RxD* 上进入的比特流与本地位时钟(*Bclk*)是不同步的。如果我们要在 *Bclk* 上升沿对 *RxD* 进行读操作，则可能存在建立和保持时间问题。如果收到信号的比特率与 *Bclk* 有微小不同，则我们就可能在错误的时间读取一些位。为了避免这个问题，我们在每个比特时间内对 *RxD* 采样 8 次（有些系统要每比特采样 16 次）。我们在时钟 *BclkX*8 的每个上升沿采样。图 11.22 中的箭头就表示时钟 *BclkX*8 上升沿。为了保证最大可行度，在理想情况下我们将在每比特时间中间进行采样。首先，当 *RxD* 变为 0 时，我们将等待 4 个 *BclkX*8 时钟周期，此时我们应该在起始位中间附近。然后，再等待 8 个 *BclkX*8 时钟周期，此时我们应该在第一个数据位中间附近。此后，每隔 8 个 *BclkX*8 时钟周期读取一次，直到停止位读取完毕。

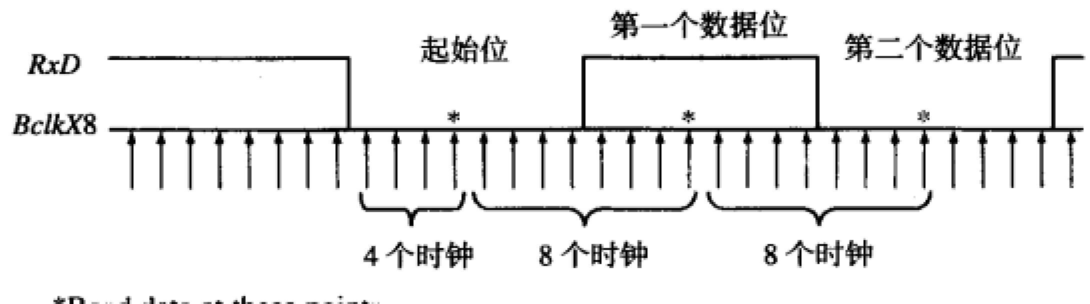


图 11.22 用 *BclkX*8 时钟对 *RxD* 采样

UART 接收机的 SM 图如图 11.23 所示。图中我们使用了两个计数器：*Ct*1 记录 *BclkX*8 时钟个数，*Ct*2 记录从起始位起接收到的比特数。在 IDLE 状态，一直等待开始位(*RxD* = ‘0’)到来，并进入到“开始位检测到”状态。在时钟 *BclkX*8 上升沿到来时（ $BClkX8\uparrow$ ），再次对 *RxD* 进行采样。由于起始位（为‘0’）要持续 8 个 *BclkX*8 时钟，所以可以读取‘0’。由于 *Ct*1 仍旧为 0，所以 *Ct*1 加 1，同时 SM 等待下一个 $BclkX8\uparrow$ 的到来。如果 *RxD* = ‘1’（这是一个错误条件），则 *Ct*1 清零并且 SM 重新回到 IDLE 状态。否则，SM 继续循环。当 *RxD* 第 4 次为‘0’时，*Ct*1 = 3，所以 *Ct*1 清零并且 SM 变为接收数据状态。在这一状态下，在每个时钟 *BclkX*8 上升沿到来时，SM 对 *Ct*1 加 1。8 个时钟过后，*Ct*1 = 7，此时检测 *Ct*2 的值。如果 *Ct*2 不为 8，则 *RxD* 的当前值被移入到 *RSR* 中，并且 *Ct*2 加 1，*Ct*1 清零。如果 *Ct*2 = 8，则所有 8 位数据都已经被读取，并且我们应该位于停止位的中间位置。如果 *RDRF* = ‘1’，则微控制器还没有读取前一个数据字节，出现了超限(overrun)错误，此时状态寄存器中的 *OE* 标志位置 1，且忽略新数据。如果 *RxD* = ‘0’，则停止位未被检测到，并且状态寄存器中的 *FE* 标志位置 1。如果没有错误，则 *RSR* 中的数据被载入到 *RDR* 中。在所有这些情况下，如果 *RDRF* 标志位置 1，则认为接收操作完成，并且把计数器清零。

UART 接收机的 VHDL 代码（参见图 11.24）是基于图 11.23 所示的 SM 图编写的。接收机包括 *RDR* 寄存器、*RSR* 寄存器和接收机控制器。控制器具有 *SCSR* 接口。*RDR* 可以把数据驱动到数据总线上。第一个进程表示一个生成下一状态信号和控制信号的组合逻辑网络。第二个进程在时钟上升沿对寄存器进行更新。在时钟 *BclkX*8 上升沿到来时，信号 *BclkX8_rising* 变为‘1’，并持续一个系统时钟周期。*BckX8_rising* 同 *BClk_rising* 的生成过程相同。

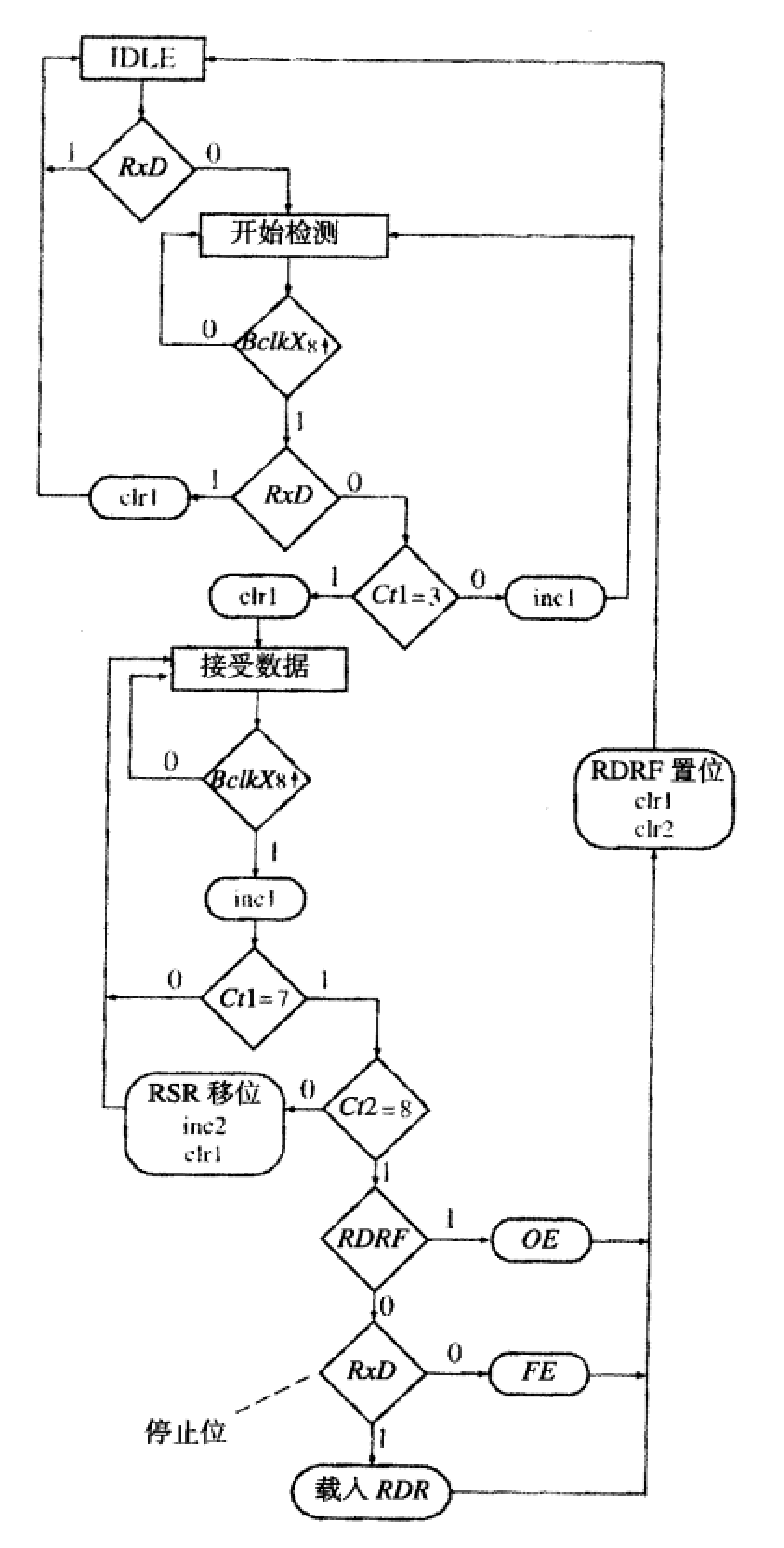


图 11.23　UART 接收机 SM 图

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all; -- to use unsigned type

entity UART_Receiver is
port(RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;
     RDR: out unsigned(7 downto 0);
     setRDRF, setOE, setFE: out std_logic);
end UART_Receiver;

architecture rcvr of UART_Receiver is
  type stateType is (IDLE, START_DETECTED, RECV_DATA);
  signal state, nextstate: stateType;
  signal RSR: unsigned(7 downto 0);   -- receive shift register
  signal ct1 : integer range 0 to 7;   -- indicates when to read the RxD input
  signal ct2 : integer range 0 to 8;   -- counts number of bits read
  signal inc1, inc2, clr1, clr2, shftRSR, loadRDR: std_logic;
```

图 11.24　UART 接收机的 VHDL 代码

```
  signal BclkX8_Dlayed, BclkX8_rising: std_logic;
begin
  BclkX8_rising <= BclkX8 and (not BclkX8_Dlayed);
    -- indicates the rising edge of bitX8 clock
  Rcvr_Control: process(state, RxD, RDRF, ct1, ct2, BclkX8_rising)
  begin
    -- reset control signals
    inc1 <= '0'; inc2 <= '0'; clr1 <= '0'; clr2 <= '0';
    shftRSR <= '0'; loadRDR <= '0'; setRDRF <= '0'; setOE <= '0'; setFE <= '0';
    case state is
      when IDLE =>
        if (RxD = '0') then nextstate <= START_DETECTED;
        else nextstate <= IDLE;
        end if;
      when START_DETECTED =>
        if (BclkX8_rising = '0') then nextstate <= START_DETECTED;
        elsif (RxD = '1') then clr1 <= '1'; nextstate <= IDLE;
        elsif (ct1 = 3) then clr1 <= '1'; nextstate <= RECV_DATA;
        else inc1 <= '1'; nextstate <= START_DETECTED;
        end if;
      when RECV_DATA =>
        if (BclkX8_rising = '0') then nextstate <= RECV_DATA;
        else inc1 <= '1';
          if (ct1 /= 7) then nextstate <= RECV_DATA;
            -- wait for 8 clock cycles
          elsif (ct2 /= 8) then
            shftRSR <= '1'; inc2 <= '1'; clr1 <= '1'; -- read next data bit
            nextstate <= RECV_DATA;
          else
            nextstate <= IDLE;
            setRDRF <= '1'; clr1 <= '1'; clr2 <= '1';
            if (RDRF = '1') then setOE <= '1';       -- overrun error
            elsif (RxD = '0') then setFE <= '1';     -- framing error
            else loadRDR <= '1';                     -- load recv data register
            end if;
          end if;
        end if;
    end case;
  end process;

  Rcvr_update: process(sysclk, rst_b)
  begin
    if (rst_b = '0') then state <= IDLE; BclkX8_Dlayed <= '0';
      ct1 <= 0; ct2 <= 0;
    elsif (sysclk'event and sysclk = '1') then
      state <= nextstate;
      if (clr1 = '1') then ct1 <= 0; elsif (inc1 = '1') then
        ct1 <= ct1 + 1;
      end if;
      if (clr2 = '1') then ct2 <= 0; elsif (inc2 = '1') then
        ct2 <= ct2 + 1;
      end if;
      if (shftRSR = '1') then RSR <= RxD & RSR(7 downto 1);
      end if;
      -- update shift reg.
      if (loadRDR = '1') then RDR <= RSR;
      end if;
      BclkX8_Dlayed <= BclkX8;    -- BclkX8 delayed by 1 sysclk
    end if;
  end process;
end rcvr;
```

图 11.24（续） UART 接收机的 VHDL 代码

图 11.25 给出了基于 Xilinx Spartan 3 FPGA 器件对 UART 接收机进行综合的结果。综合后的电路需要 26 个触发器、21 个片和 32 个 4 输入 LUT。

下面我们设计一个可编程波特率生成器。*SCCR* 中的三个位用于对 8 种波特率中的一种进行

选择。假设系统时钟为 8 MHz，我们需要的波特率为 300, 600, 1200, 2400, 4800, 9600 ,19 200 和 38 400。所需的 *BclkX*8 的最大频率为 38 400×8 = 307 200 Hz。为了得到该频率，我们应该把 8 MHz 除以 26.04。因为我们只能用整数除数，所以我们要么接受一个有小误差的波特率，要么把系统时钟变为 7.9877 MHz 以适应波特率的需要。

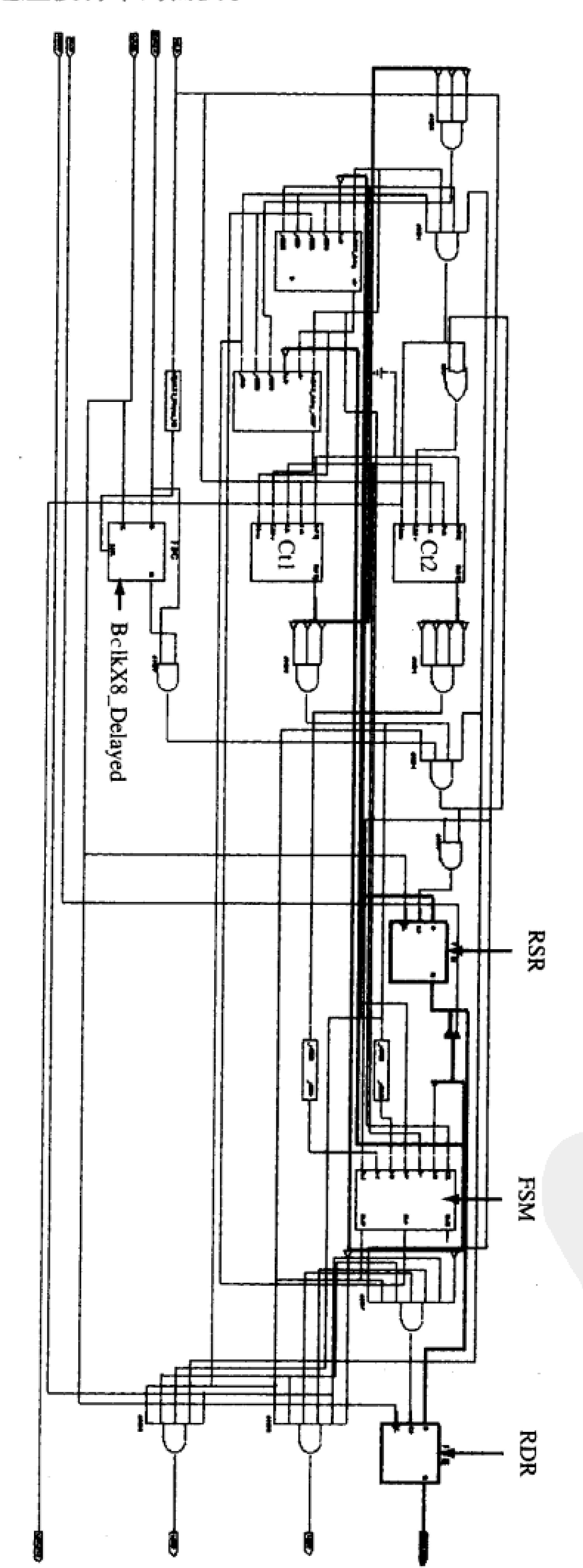
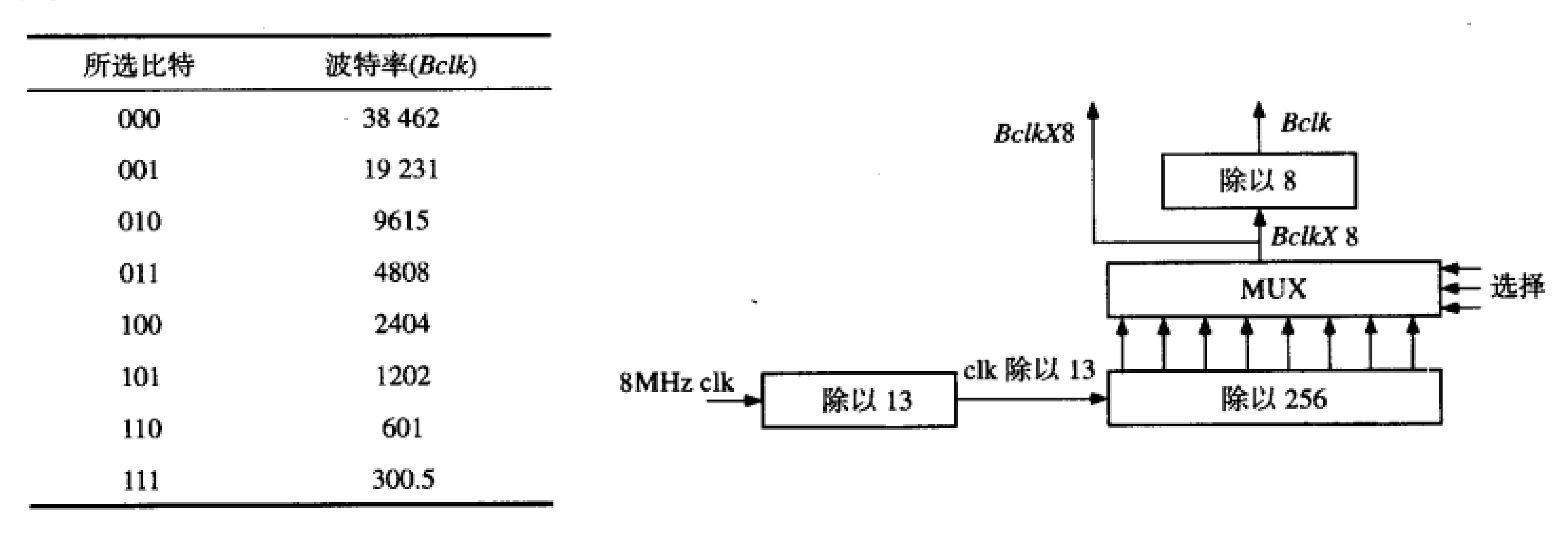


图 11.25 综合的 UART 接收机

波特率生成器的框图如图 11.26 所示。首先使用计数器把 8 MHz 的系统时钟除以 13。此计数器的输出传输到一个 8 位二进制计数器，且其触发器的输出相应地除以 2, 4, …, 256，多路选择器从这些输出中选择一个。多路选择器的输入为 *SCCR* 的低三位，且其输出与 *BclkX*8 相对应，所以应再除以 8，使之与 *Bclk* 相对应。假设现在有一个 8 MHz 的时钟，则可以生成的频率如下所示。



所选比特	波特率(*Bclk*)
000	38 462
001	19 231
010	9615
011	4808
100	2404
101	1202
110	601
111	300.5

图 11.26　波特率生成器

波特率生成器的 VHDL 代码如图 11.27 所示。代码中第一个进程在系统时钟上升沿到来时，对模 13 计数器加 1。第二个进程在 *Clkdiv*13 上升沿到来时，对模 256 计数器加 1。用并发语句生成 MUX 的输出(*BclkX*8)。第三个进程在 *BclkX*8 上升沿到来时，对模 8 计数器加 1，并生成 *Bclk*。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;    -- for overloaded + operator and conversion functions

entity clk_divider is
  port(Sysclk, rst_b: in std_logic;
       Sel: in unsigned(2 downto 0);
       BclkX8: buffer std_logic;
       Bclk: out std_logic);
end clk_divider;

architecture baudgen of clk_divider is
  signal ctr1: unsigned(3 downto 0) := "0000";     -- divide by 13 counter
  signal ctr2: unsigned(7 downto 0) := "00000000";    -- div by 256 ctr
  signal ctr3: unsigned(2 downto 0) := "000";    -- divide by 8 counter
  signal Clkdiv13: std_logic;
begin
  process(Sysclk)    -- first divide system clock by 13
  begin
    if (Sysclk'event and Sysclk = '1') then
      if (ctr1 = "1100") then ctr1 <= "0000";
      else ctr1 <= ctr1 + 1;
      end if;
    end if;
  end process;
  Clkdiv13 <= ctr1(3);     -- divide Sysclk by 13

  process(Clkdiv13)    -- ctr2 is an 8-bit counter
  begin
    if (Clkdiv13'event and Clkdiv13 = '1') then
      ctr2 <= ctr2 + 1;
    end if;
  end process;

  BclkX8 <= ctr2(to_integer(sel));    -- select baud rate
```

图 11.27　波特率生成器的 VHDL 代码

```
  process(BclkX8)
  begin
    if (BclkX8'event and BclkX8 = '1') then
      ctr3 <= ctr3 + 1;
    end if;
  end process;
  Bclk <= ctr3(2);     -- Bclk is BclkX8 divided by 8
end baudgen;
```

图 11.27（续）　波特率生成器的 VHDL 代码

为了完成 UART 的设计，我们需要把之前设计的三个组成元件组合起来，把它们同控制和状态寄存器连接起来，并加入中断生成逻辑和总线接口。UART 的完整 VHDL 代码如图 11.28 所示。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity UART is
  port(SCI_sel, R_W, clk, rst_b, RxD: in std_logic;
       ADDR2: in unsigned(1 downto 0);
       DBUS: inout unsigned(7 downto 0);
       SCI_IRQ, TxD: out std_logic);
end UART;

architecture uart1 of UART is
  component UART_Receiver
    port(RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;
         RDR: out unsigned(7 downto 0);
         setRDRF, setOE, setFE: out std_logic);
  end component;
  component UART_Transmitter
    port(Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
         DBUS: in unsigned(7 downto 0);
         setTDRE, TxD: out std_logic);
  end component;
  component clk_divider
    port(Sysclk, rst_b: in std_logic;
         Sel: in unsigned(2 downto 0);
         BclkX8: buffer std_logic; Bclk: out std_logic);
  end component;
  signal RDR: unsigned(7 downto 0);   -- Receive Data Register
  signal SCSR: unsigned(7 downto 0);   -- Status Register
  signal SCCR: unsigned(7 downto 0);   -- Control Register
  signal TDRE, RDRF, OE, FE, TIE, RIE: std_logic;
  signal BaudSel: unsigned(2 downto 0);
  signal setTDRE, setRDRF, setOE, setFE, loadTDR, loadSCCR: std_logic;
  signal clrRDRF, Bclk, BclkX8, SCI_Read, SCI_Write: std_logic;
begin
  RCVR: UART_Receiver port map(RxD, BclkX8, clk, rst_b, RDRF, RDR,
                               setRDRF, setOE, setFE);
  XMIT: UART_Transmitter port map(Bclk, clk, rst_b, TDRE, loadTDR,
                                  DBUS, setTDRE, TxD);
  CLKDIV: clk_divider port map(clk, rst_b, BaudSel, BclkX8, Bclk);
  -- This process updates the control and status registers
  process(clk, rst_b)
  begin
    if (rst_b='0') then
      TDRE <= '1'; RDRF <= '0'; OE <= '0'; FE <= '0';
      TIE <= '0'; RIE <= '0';
    elsif (rising_edge(clk)) then
      TDRE <= (setTDRE and not TDRE) or (not loadTDR and TDRE);
      RDRF <= (setRDRF and not RDRF) or (not clrRDRF and RDRF);
      OE <= (setOE and not OE) or (not clrRDRF and OE);
      FE <= (setFE and not FE) or (not clrRDRF and FE);
      if (loadSCCR = '1') then TIE <= DBUS(7); RIE <= DBUS(6);
        BaudSel <= DBUS(2 downto 0);
      end if;
```

图 11.28　UART 的完整 VHDL 代码

```
    end if;
  end process;

  -- IRQ generation logic
  SCI_IRQ <= '1' when ((RIE='1' and (RDRF='1' or OE='1')) or
                       (TIE='1' and TDRE='1'))
                 else '0';
  -- Bus Interface
  SCSR <= TDRE & RDRF & "0000" & OE & FE;
  SCCR <= TIE & RIE & "000" & BaudSel;
  SCI_Read <= '1' when (SCI_sel = '1' and R_W = '0') else '0';
  SCI_Write <= '1' when (SCI_sel = '1' and R_W = '1') else '0';
  clrRDRF <= '1' when (SCI_Read = '1' and ADDR2 = "00") else '0';
  loadTDR <= '1' when (SCI_Write = '1' and ADDR2 = "00") else '0';
  loadSCCR <= '1' when (SCI_Write = '1' and ADDR2 = "10") else '0';
  DBUS <= "ZZZZZZZZ" when (SCI_Read = '0')   -- tristate bus when not reading
          else RDR when (ADDR2 = "00")
  -- write appropriate register to the bus
          else SCSR when (ADDR2 = "01")
          else SCCR;    -- dbus = sccr, if ADDR2 is "10" or "11"
end uart1;
```

图 11.28（续） UART 的完整 VHDL 代码

SCI_IRQ 是中断信号。当我们需要关注 UART 接收机和发送机时，它对 CPU 产生中断。当 *SCCR* 中的标志位 *RIE* 置 1 时，无论 *RDRF* 为 1 或 *OE* 为'1'，*SCI_IRQ* 都会被生成。当 *SCCR* 中的标志位 *TIE* 置 1 时，无论 *TDRE* 是否为'1'，SCI_IRQ 都会被生成。

UART 与微控制器地址和数据总线是有接口的，所以当 *SCIsel* = '1'时，CPU 可以对 UART 寄存器进行读写操作。地址(*ADDR*2)的最后 2 位，与 *R_W* 信号都用于寄存器的选择。

ADDR2	*R_W*	操作
00	0	*DBUS* ← *RDR*
00	1	*TDR* ← *DBUS*
01	0	*DBUS* ← *SCSR*
01	1	*DBUS* ← 高阻
1–	0	*DBUS* ← *SCCR*
1–	1	*SCCR* ← *DBUS*

当 UART 未被选中为读操作时，数据总线为高阻态。

若图 11.28 所示 VHDL 代码用 Xilinx Spartan 3 系列 FPGA 器件进行综合，则综合后的电路需要 74 个触发器、62 个片和 109 个 4 输入 LUT。

本章中说明了如何用 VHDL 语言实现数字系统的设计与仿真，并介绍了三个实例：两个设计实例为多功能手表和 UART，一个仿真实例为存储芯片的读取。在设计实例中，我们先给出设计的框图和系统控制器的 SM 图，然后给出该系统中各个模块的行为描述 VHDL 代码。我们也说明了测试的用法。最后，针对 FPGA 综合了该 VHDL 代码，并把设计下载到开发板上验证了操作功能。

本章中还介绍了一个存储器芯片的仿真模型。这一模型包含了存储器的时间参数和内嵌测试，用于测试建立间、保持时间和其他的时序规范。当基于第三方内核/芯片设计片上系统(SoC)时，这种模型是很有帮助的。

习题

11.1 假设要在一个 FPGA 板上实现 11.1 节中介绍的手表系统。针对你有的 FPGA 板，为手表系统设计一个输入模块，并写出 VHDL 代码。

11.2 假设要在一个 FPGA 板上实现 11.1 节中介绍的手表系统。设计手表的显示模块并写出 VHDL

代码。根据手表的不同模式，使用 FPGA 板上的 LCD 显示时间、闹钟设定或秒表。

11.3 (a) 在图 11.2 和图 11.3 的手表模块上加入一个倒计时模块。该模块可以对小时、分和秒计时。在计时状态，当按下 B2 时，我们应该可以使用 B3 设定小时、分和秒。当设定完成时，手表回到主计时状态，按下 B3 开始倒计时。如果 B3 被再次按下，则倒计时停止，否则，当变为 00:00:00 时，倒计时停止，并且计数器在最后一秒发出蜂鸣声。

(b) 对测试程序进行修改，并对该倒计时器进行检测。

11.4 下面我们设计一个简单的计算器，它可以计算无符号二进制数加法。该计算器的操作同普通简易计算器大体相同，只是所有的输入均为二进 制数，而且只可以计算'+'操作。此计算器可以显示带小数点的二进制 8 位数，而且只有 5 个按键：0, 1,+和复位键。按下复位键，则所有的寄存器清零，且计算器回到开始状态。在键入第一个数后，按下+键，然后再键入第二个数，此操作可以重复进行直到按下复位键为止。注意这里未使用等号键。假设我们输入了一个正常的序列，也就是说，在按+键前总有一个二进制数被键入。在加法操作进行完毕前，二进制数的小数点必须通过移位进行校正。如果产生溢出，则尽可能地对其进行校正。如果校正失败，则把 *E* 置 1，报错。

按键不必进行编码。计算器有 6 个输入信号：*zero. one. dot, plus, reset* 和 *V*。假设在一个时钟内任何时刻按键时，所有的输入信号均发生抖动，且 *V*=1。显示输出为 8 位，来自 *A* 寄存器、*RCTA*（二进制小数点右边的位数）和 *E*。

(a) 画出该计算器的实现框图，要求显示所需的寄存器、计数器、加法器等。给出必要的控制 信号，并对其含义加以解释。例如，RSHA 是指对 A 进行右移。给出每个寄存器的具体大小。

(b) 画出计算器主要程序的 SM 图。要求包括输入二进制数，校正二进制小数点，加法操作，溢出校正。对所有使用的控制信号加以定义。

(c) 写出计算器主要模块的 VHDL 代码。

(d) 给出你的 VHDL 模块的测试平台。

11.5 观察静态 RAM 的读写周期(参见图 11.11 和图 11.12),回答下列问题(不用给出具体数值说明)。

(a) 如果 $\overline{WE}$ = 1，且在 $\overline{CS}$ 变为 0 的同时，地址改变，则 RAM 输出端有效数据可用前最大时间为多少？（注意：时序图不是按比例画出的）

(b) 从 RAM 中每秒可读取的最大字节数是由什么决定的？列出你能想到的所有可能性。

(c) 对于由 $\overline{WE}$ 控制的写周期，在写入 RAM 时，正常的操作顺序是什么？

(d) 为了能够正确地把数据写入 RAM 中，列出必须要满足的时序条件。例如，至少 t_{WP} 时间内 $\overline{WE}$ 为 0。

11.6 回答下列关于 6116 SA-15 静态 CMOS RAM 的问题（基于表 11.1）。

(a) 可用的最大时钟频率为多少？

(b) 求当地址或 $\overline{CS}$ 发生变化时，有效数据可读取的最小时间。

(c) 求在一个 $\overline{WE}$ 控制的写周期中，当 $\overline{WE}$ 变为低电平后，新数据可被载入的最早时间。

(d) 求在一个 $\overline{CS}$ 控制的写周期中，当地址发生变化后，有效数据可被载入的最小时间。

11.7 6116 CMOS RAM 简易存储模块中，假设 $\overline{CS}$ 和 $\overline{OE}$ 始终为低电平，所以存储操作只与地址和 $\overline{WE}$ 有关。

(a) 忽略所有的时间信号，写出模块的 VHDL 代码（代码中不包含 $\overline{CS}$ 和 $\overline{OE}$ ）。

(b) 在代码中加入时间 t_{AA}，t_{OH}，t_{WHZ}，t_{OW}。读操作时，*Dout* 先在 t_{OH} 之后变为"XXXXXXXX"

（未知），然后在 t_{AA} 后输出有效数据。写操作时，*Dout* 先在 t_{WHZ} 之后变为高阻，然后在 t_{OW} 后变为"XXXXXXXX"。

(c) 再添加一个进程，给出不满足下列时间信号时的错误信息：t_{WP}，t_{DH}，t_{DW}。

11.8 图 11.14 给出了 6116 存储器工作过程的 VHDL 代码。

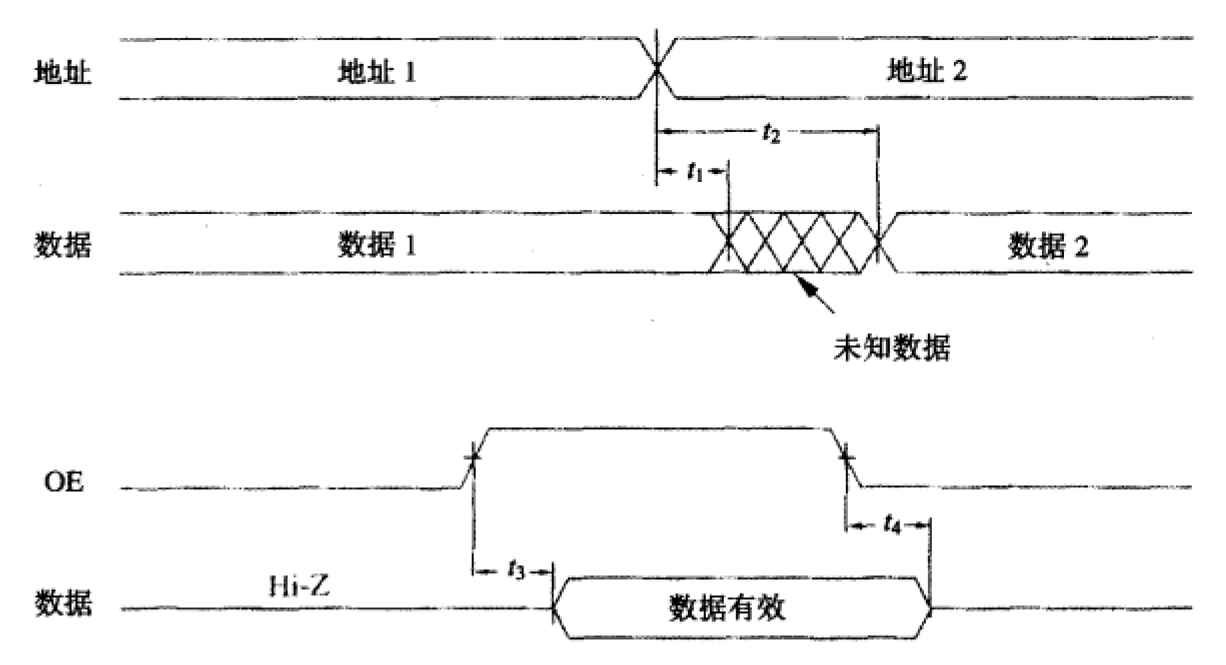
(a) 当数据写入存储器时建立时间不满足，或保持时间不满足，或 *WE_b* 的最小脉冲宽度不满足时，试说明该代码会报错。

(b) 当考虑 *OE_b*($\overline{OE}$)时，指出在原代码中要修改的部分和添加的部分。注意：读操作时，如果 *OE_b* 在 *CS_b* 变为低电平后也变为低电平，则必须要考虑进入时间 t_{OE}；如果 *OE_b* 变为高电平，数据总线在时间 t_{OHZ} 后变为高阻态。

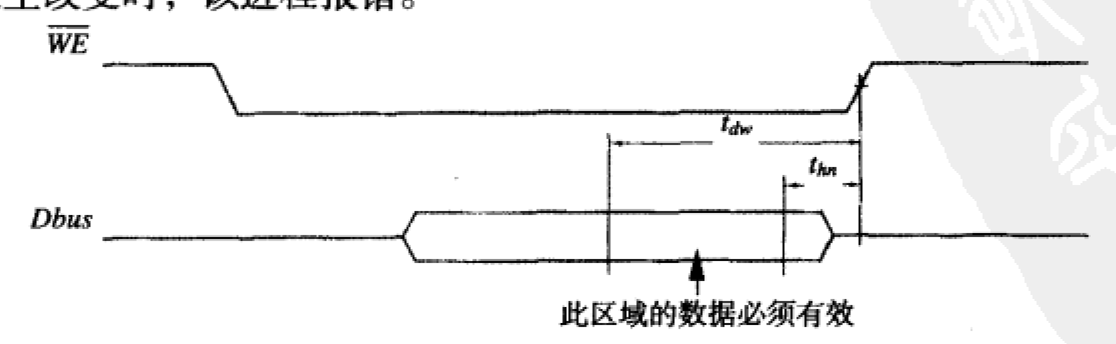
11.9 为了验证 6116 RAM VHDL 时间模型（参见图 11.14）的建立时间(t_{AS})和写恢复时间(t_{WR})的规范要求，需要该模型中的检测进程做何改动？

11.10 对于静态 CMOS RAM 中由 *CS* 控制的写周期（见图 11.13），为了验证 *CS* 控制的写操作，应对图 9.11 中的检测进程做何改动？你必须检测其时序规范，如 t_{CW}，t_{DH}，t_{DW}。

11.11 一个 ROM 具有 8 位地址输入、一个输出使能(*OE*)和一个 8 位数据输出。当 *OE* = 0 时，*data*= Hi-Z；当 *OE* = 1 时，从 ROM 读取数据。时序图如下所示。写出满足如下时序的 ROM 的 VHDL 代码。



11.12 一个由 $\overline{WE}$ 控制的静态 RAM 存储器的写周期如下图所示。该存储器具有长度为 t_{hn} 的负数据保持时间。这就意味着只要满则建立时间(t_{dw})，则输入数据可以在 $\overline{WE}$ 上升沿前 t_{hn} 时间内任意时刻发生改变。编写一个进程，当输入数据(*Dbus*)在 $\overline{WE}$ 上升沿前 t_{dw} 到 t_{hn} 之间的任意时刻发生改变时，该进程报错。



11.13 对 UART 接收机 VHDL 代码做必要的修改，使其能够使用 16X 位的时钟，以代替 8X 位的时钟。使用更快的时钟可以增强接收机的抗噪能力。

11.14 (a) 为 UART 编写一个 VHDL 测试平台。要求包括对超限错误、帧错误、有噪声导致的启动失败、BAUD 率的改变等。对该 VHDL 代码进行仿真。

(b) 如果有可用的硬件，写出一个更简单的测试平台，对 *TxD* 与 *RxD* 的外部连接进行自闭合测试。综合测试平台和 UART 模块，并下载到目标器件上，验证硬件操作。

11.15 对 11.3 节中 UART 代码做必要的修改，使其具有校验功能。在 *SCCR* 中加入两位(P_1P_0)用来选择校验模式，可选的模式如下所示：

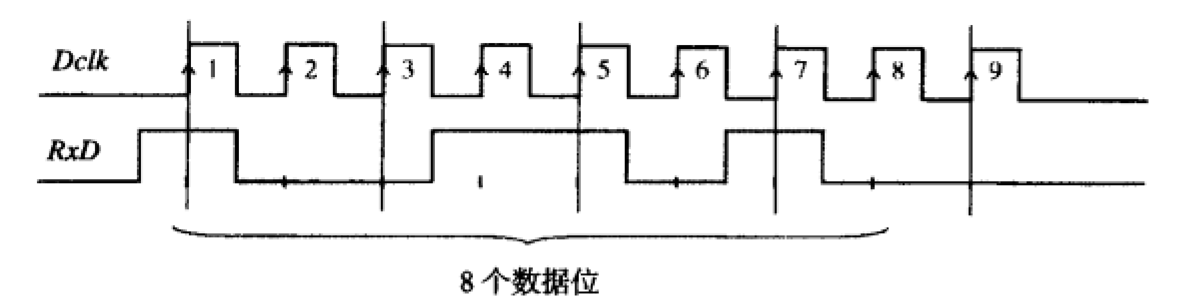
$P_1P_0=00$　8 个数据位，无校验位
$P_1P_0=01$　7 个数据位，第 8 位为偶校验位
$P_1P_0=00$　7 个数据位，第 8 位为奇校验位
$P_1P_0=11$　7 个数据位，第 8 位总为'0'

发送机应生成上述偶、奇和 0 校验位。接收机应对校验位进行检测并验证其正确与否。如果不正确，则把 *SCSR* 中的 *PE* 标志位置 1。

11.16 有一个同步接收机，其操作与 11.3 节中介绍的 UART 接收机有相似之处，只是该同步接收机可以同时发送数据 *RxD* 和数据时钟 *Dclk*，这样就不用使用本地时钟对数据进行同步，也就无须起始位和停止位。如下所示，当发送 8 位数据时，时钟在 9 个时钟周期内有效，然后变为无效。头 8 个时钟数据移入到接收移位寄存器(*RSR*)中，在第 9 个时钟，数据转移到接收数据寄存器(*RDR*)，且 *RDRF* 标志位置 1。

(a) 画出此同步接收机的实现框图，要求包括一个计数器（注意，无须使用状态机，但是需要生成控制信号 *Load* 和 *Shift*）。

(b) 写出与(a)相对应的 VHDL 代码。信号 *Load* 和 *Shift* 必须明确出现在你的代码中。



11.17 写出 UART 的测试平台，能完成对 UART 的自闭合检测。测试平台把 UART 的输出 *TxD* 和输入 *RxD* 连接起来，所以任何被载入到 *TDR* 的数据都将自动发送到 *TxD* 中，并被接收到 *RxD* 中，然后再载入到 *RDR* 中。此测试平台必须能够模拟如下操作：CPU 在 *TDR* 中写入“01010101”；通过一个循环不断对状态寄存器进行读操作，直到 *RDRF* = '1'；然后再从 *RDR* 中读取数据。

附录 A VHDL 语言小结

声明：这个 VHDL 的小结是不完全的，并且包含某些特殊的情况。该小结只列出了在本文中出现过的 VHDL 语句。完整的 VHDL 语法请参阅参考文献[6, 9, 23]。

注意：

- VHDL 中不区分大小写。
- 信号名和其他标识符可以包含字母、数字和下划线(_)。
- 标识符必须以字母开头。
- 标识符不能以下划线结尾。
- 每条 VHDL 语句都必须以分号结尾。
- VHDL 对类型要求很严格。通常不允许数据类型的混用。

图例

黑体	保留字
[ ]	可选项
{ }	可重复 0 次或者多次
|	或者

1. 预定义类型

位(bit)	逻辑 0 或 1。
布尔量	逻辑假或真。
整数	整数，取值范围为$-(2^{31}-1)$～$+(2^{31}-1)$，在某些实现中取值范围更宽。
实数	浮点数，取值范围为-1.0E38～+1.0E38。
字符	包含上划线、小写字母、数字和特殊符号在内的任何合法 VHDL 字符（每个可显字符必须用单引号括起来，例如'd','7','+'）。
时间	带单位的整数，单位为 fs,ps,ns,us,ms,sec,min,hr。
自然数	大于等于 0 的整数。
正数	大于 0 的整数。
位矢量(bit_vector)	位数组。
字符串(string)	字符数组。
延迟长度	时间≥0。

2. 运算符（按优先级从小到大排列）

（1）二进制逻辑运算符：　**and or nand nor xor xnor**

（2）关系运算符：　= /= < <= > >=

（3）移位运算符：　**sll srl sla sra rol ror**

（4）加法运算符：　+ – &（拼接）

（5）正负运算符:　+ −

（6）乘法运算符:　*** / mod rem**

（7）辅助运算符:　**not abs ****

3. 预定义属性

信号属性（返回一个值）:

属　性	返 回 值
S' ACTIVE	若信号在当前 Δ 时间内发生事务，则返回真，否则返回假
S' EVENT	若信号在当前 Δ 时间内发生事件，则返回真，否则返回假
S' LAST_EVENT	S 的上一次事件发生到目前的时间差
S' LAST_VALUE	上一次事件发生之前的 S 值
S' LAST_ACTIVE	S 的从上一次事务发生到目前的时间差

生成信号的信号属性:

属　性	生成的信号
S' DELAYED[(time)]*	与 S 同类型的信号，但是比信号 S 延迟指定的时间
S' STABLE[(time)]*	布尔信号（若 S 在指定时间内无事件发生，则布尔信号为真）
S' QUIET[(time)]*	布尔信号（若 S 在指定时间内无事务发生，则布尔信号为真）
S' TRANSACTION	bit 类型的信号。每当 S 发生事务时，此信号就发生改变

* 当没有指定时间时，时间为 Δ 。

数组属性:

```
type ROM is array (0 to 15,7 downto 0) of bit; signal ROM1: ROM;
```

属　性	返　回	示　例
A' LEFT(N)	第 N 个下标取值范围的左边界	ROM1' LEFT(1) = 0
		ROM1' LEFT(2) = 7
A' RIGHT(N)	第 N 个下标取值范围的右边界	ROM1' RIGHT(1) = 15
		ROM1' RIGHT(2) = 0
A' HIGH(N)	第 N 个下标取值范围的最大边界	ROM1' HIGH(1) = 15
		ROM1' HIGH(2) = 7
A' LOW(N)	第 N 个下标取值范围的最小边界	ROM1' LOW(1) = 0
		ROM1' LOW(2) = 0
A' RANGE(N)	第 N 个下标取值范围	ROM1' RANGE(1) = 0 to 15
		ROM1' RANGE(2) = 7 downto 0
A' REVERSE_RANGE(N)	第 N 个上标取值范围的边界反序	ROM1' REVERSE_RANGE(1) = 15 downto 0
A' LENGTH(N)	第 N 个下标取值范围的大小	ROM1' REVERSE_RANGE(2) = 0 to 7
		ROM1' LENGTH(2) = 8

4. 预定义函数

NOW　返回当前仿真时间

FILE_OPEN([状态]，文件号，字符串，端口传输模式)　打开文件

FILE_CLOSE(文件号)　　　　关闭文件

5. 说明语句

实体说明：

```
entity 实体名 is
    [generic （类属列表及其数据类型）；]
    [port（端口信号说明）；]
    [说明语句]
end [entity] [实体名]；
```

端口信号说明：

```
端口信号列表：模式 '数据类型[:= 初始值]
{；端口信号列表：模式 '数据类型[:= 初始值]}
```

注意：端口信号的模式可以是 in, out, inout 和 buffer。

结构说明：

```
architecture 结构体名 of 实体名 is
        [定义语句]
begin
        结构体主体
end [architecture] [结构体名]；
```

注意：结构体可以包含元件例化语句、进程、赋值语句、过程调用等。

整数类型说明：

```
type 类型名 is range 整数取值范围；
```

枚举类型说明：

```
type 类型名 is 名称或字符列表；
```

子类型说明：

```
subtype 子类型名 is 类型名 [下标或取值范围限制]；
```

变量说明：

```
variable 变量名列表：类型名[:= 初始值]；
```

信号说明：

```
signal 信号名列表：类型名[:= 初始值]；
```

常数说明：

```
constant 常数名：类型名：= 常数值；
```

别名说明：

```
alias 标识符[：标识符类型] is 项目名称；
```

注意：项目名称可以是一个常数、信号、变量、文件、类型名等。

过程说明：

```
procedure 过程名（参数列表） is
        [定义语句]
begin
        顺序语句
end 过程名；
```

注意：参数可以是信号、变量或常数。

函数说明：

```
function 函数名 （参数列表） return 返回类型 is
          [定义语句]
begin
          顺序语句          --必须包括return返回值；
end 函数名；
```

注意：参数可以是信号或常数。

库说明：

```
library 库名列表；
```

use 语句：

```
use 库名.包集合名.项目；    （.项目可以为.all）
```

包集合说明：

```
package 包集合名 is
        包集合说明
end [package] [包集合名]；
```

包集合主体：

```
package body 包集合名 is
        包集合主体说明
end [package body] [包集合名]；
```

元件说明：

```
component 元件名
      [generic （类属列表及其数据类型）；]
      port （端口信号列表及其数据类型）；
end component；
```

文件类型说明：

```
type 文件名 is file of 类型名；
```

文件说明：

```
file 文件名：文件类型 [open模式] is "文件路径名"；
```

注意：在这里模式指读模式、写模式或附加模式。

6. 并发语句

信号赋值语句：

```
signal <= [reject 脉冲宽度 | transport] 表达式 [ after 延迟时间]；
```

注意：如果信号赋值是并发的，在每次式子右边发生变化的时候，都会重新计算信号的值。

如果在 **after** 后没有延迟时间，信号在Δ时间之后更新。

条件赋值语句：

```
signal <= 表达式1 when 条件1
          else 表达式2  when 条件2
          ...
         [else 表达式]；
```

选择信号赋值语句：

```
with 表达式 select
     signal <= 表达式1  [after 延迟1] when 条件1，
               表达式2  [after 延迟2] when 条件2，
               ...
              [表达式 [after 延迟]when others]；
```

断言语句：

```
assert 布尔表达式
       [report 数据串表达式]
       [severity 严重级别]；
```

元件例化：

```
标识：元件名
     [generic map (类属的关联列表)；]
     port map (实际信号列表)；
```

注意：如果元件的输出没有连接，则使用 **open**。

生成语句：

```
生成标识 ： for 标识符 in 范围 generate
[begin]
    并发语句
end generate [生成标识]；

生成标识 ： if 条件 generate
[begin]
    并发语句
end generate [生成标识]；
```

进程语句（带有敏感信号表）：

```
[进程标识:] process (敏感信号表)
        [说明语句]  -- 不允许有信号的说明
begin
        顺序语句
end process [进程标识];
```

注意：这种形式的进程在初始化时执行，之后只有当敏感表中的任一项发生变化之后它才被执行。敏感表是一个信号的列表。不能有 wait 语句。

进程语句（不带敏感信号表）：

```
[进程标识:] process
        [说明语句]  -- 不允许有信号的说明
begin
        顺序语句
end process [进程标识];
```

注意：这种类型的进程必须至少含有一个 wait 语句。进程会立刻执行并且一直持续，直到遇到一条 wait 语句。

过程调用：

```
过程名 (实参变量列表);
```

注意：当实参变量的模式为 in 时，实参变量可以使用一个表达式；实参变量的数据类型必须与形参变量的数据类型相匹配；不可以使用 open。

函数调用：

```
函数名(实参变量列表)
```

注意：函数调用可以在表达式中使用，也可以代替表达式使用。函数调用本身并不是一条语句，它只是语句的一部分。

7. 顺序语句

信号赋值语句：

```
signal <= [reject 脉冲宽度 | transport] 表达式 [ after 延迟时间];
```

注意：如果省略[after 延迟时间]，那么信号在 Δ 时间之后更新。

变量赋值语句：

```
variable := 表达式;
```

注意：此语句只能在进程、函数或过程中使用。变量总是立即更新。

wait 语句形式如下：

```
wait on 敏感信号表;
wait until 布尔表达式;
```

```
wait for 时间表达式;
```

if 语句：

```
if 条件 then
      顺序语句
{elsif 条件 then
      顺序语句}-可包含0个或多个elsif语句
[else 顺序语句]
end if;
```

case 语句：

```
case 表达式 is
      when 选择1 => 顺序语句
      when 选择2 => 顺序语句
      …
      [when others => 顺序语句]
end case;
```

for 循环语句：

```
[循环标识:] for 指针 in 范围 loop
        顺序语句
end loop [循环标识];
```

注意：可以用 **exit** 语句跳出当前的循环。

while 循环语句：

```
[循环标识:] while 布尔表达式 loop
   顺序语句
end loop [循环标识];
```

exit 语句：

```
exit [循环标识] [when 条件];
```

assert 语句：

```
assert 布尔表达式
         [report 数据串表达式]
         [severity 严重程度];
```

report 语句：

```
report 字符串表达式
         [severity 严重程度];
```

过程调用：

```
过程名 (实参变量列表);
```

注意：当实参变量的模式为 in 时，实参变量可以使用一个表达式；实参变量的数据类型必

须与形参变量的数据类型相匹配；不可以使用 open。

函数调用：

```
函数名(实参变量列表) ;
```

注意：函数调用可以在表达式中使用，也可以代替表达式使用。函数调用本身并不是一条语句，它只是语句的一部分。



附录B IEEE 标准库

在本书中我们使用了 IEEE 库中的两个包集合——NUMERIC_BIT 和 NUMERIC_STD。这两个包集合的标头如下所示。

标准 VHDL 综合包集合（1076.3，NUMERIC_BIT）

-- 开发者：IEEE DASC 综合工作组，PAR 1076.3
-- 目的：本包集合定义了各种用于综合器中的数字类型和算术函数。本包集合中定义了两
-- 个数字类型：
-- ：--＞UNSGNED：代表一个用矢量形式表示的无符号数。
-- ：--＞SGNED：代表一个用矢量形式表示的有符号数。
-- ：基本元素类型为位(bit)。
-- ：把最左边的位作为最高有效位
-- ：有符号矢量用二进制补码表示
-- ：此包集合包含 SIGNED 和 UNSIGNED 数据类型的重载运算符。同时，此包集合还
-- ：包含很多有用的数据类型转换函数，时钟检测函数和其它实用函数。

标准 VHDL 综合包集合（1076.3，NUMERIC_STD）

-- 开发者：IEEE DASC 综合工作组，PAR 1076.3
-- 目的：本包集合定义了各种用于综合器中的数字类型和算术函数。本包集合中定义了两
-- 个数字类型：
-- ：--＞UNSGNED：代表一个用矢量形式表示的无符号数。
-- ：--＞SGNED：代表一个用矢量形式表示的有符号数。
-- ：基本元素类型为 STD_LOGIC。
-- ：把最左边的位作为最高有效位
-- ：有符号矢量用二进制补码表示
-- ：此包集合包含 SIGNED 和 UNSIGNED 数据类型的重载运算符。同时，此包集合还
-- ：包含很多有用的数据类型转换函数。

整个包集合列表见

http://www.eda.org/rassp/vhdl/models/standards/numeric_bit.vhd

http://www.eda.org/rassp/vhdl/models/standards/numeric_std.vhd

numeric_bit 包集合中很有用的转换函数

`TO_INTEGER(A)`：将一个无符号（或有符号）矢量 *A* 转换为一个整数。

`TO_UNSIGNED(B, N)`：将一个整数转换为一个长度为 *N* 的无符号矢量。

TO_SIGNED(B, N)：将一个整数转换为一个长度为 *N* 的有符号矢量。
UNSIGNED(A)：使编译器把位矢量 *A* 作为一个无符号矢量进行处理。
SIGNED(A)：使编译器把位矢量 *A* 作为一个有符号矢量进行处理。
BIT_VECTOR(B)：使编译器把一个无符号矢量 *B* 作为一个位矢量进行处理。

在 numeric_std 包集合中也有相同的转换函数，只是要用 std_logic_vector 代替 bit_vector。

注意：

1. numeric_bit 包集合含有计算一个整数和一个无符号数加法的重载运算符，但是不含有数据类型为位和无符号数的加法重载运算符。这样，若 *A* 和 *B* 均为无符号数，则可以计算 A + B + 1，但是当 carry（进位）的数据类型为位时，不允许出现下面的语句：

```
Sum <= A + B + carry;
```

   在求解无符号矢量 A+B 与 carry 的和之前，必须要把 carry 转换为无符号数。我们可以使用符号 **unsigned' (0 => carry)**完成所需的转换。因此，所用的语句为

```
Sum <= A + B + unsigned' (0 => carry);
```

2. 如果想要求得的和的位数大于加数的位数，那么必须把加数进行扩展（使用'0'进行拼接）。例如，*X* 和 *Y* 均为 4 位，如果我们使用拼接'0'的方法对 *X* 进行扩展使其变为 5 位（*Y* 会自动进行扩展匹配），那么将会得到一个 5 位的和以及进位输出。因此，

```
Sum5 <= '0' & X + Y;
```

   此语句可以实现两个 4 位数相加，并得到一个 5 位的和。



附录 C　TEXTIO 包集合

```
package TEXTIO is
-- 文本 I/O 的类型定义
-- Type definitions for text I/O:
type LINE is access STRING; -- LINE 为指向一个值为 STRING 的指针。
-- 此类预定义的运算符如下所示:
-- function "=" (anonymous, anonymous: LINE) return BOOLEAN;
-- function "/=" (anonymous, anonymous: LINE) return BOOLEAN;
type TEXT is file of STRING; -- A file of variable-length ASCII records.
-- 此类预定义的运算符如下所示:
-- procedure FILE_OPEN (file F: TEXT; External_Name; in STRING;
-- Open_Kind: in FILE_OPEN_KIND := READ_MODE);
-- procedure FILE_OPEN (Status: out FILE_OPEN_STATUS; file F: TEXT;
-- External_Name: in STRING;
-- Open_kind: in FILE_OPEN_KIND :=READ_MODE);
-- procedure FILE_CLOSE (file F: TEXT);
-- procedure READ (file F: Text; VALUE: out STRING);
-- procedure WRITE (file F: Text; VALUE: in STRING);
-- function ENDFILE (file F: TEXT) return BOOLEAN;
type SIDE is (RIGHT,LEFT); -- 使输出数据在取值范围内。
-- 此类预定义的运算符如下所示:
-- function "=" (anonymous, anonymous: SIDE) return BOOLEAN;
-- function "/=" (anonymous, anonymous: SIDE) return BOOLEAN;
-- function "<" (anonymous, anonymous: SIDE) return BOOLEAN;
-- function "<=" (anonymous, anonymous: SIDE) return BOOLEAN;
-- function ">" (anonymous, anonymous: SIDE) return BOOLEAN;
-- function "=>" (anonymous, anonymous: SIDE) return BOOLEAN;
subtype WIDTH is NATURAL; -- For specifying widths of output fields.

-- 标准文本文件:
file INPUT: TEXT open READ_MODE is "STD_INPUT".
file OUTPUT: TEXT open WRITE_MODE is "STD_INPUT".
--Input routines for standard types:
procedure READLINE (file F: TEXT; L: inout LINE);
```

```
procedure READ (L: inout LINE; VALUE: out BIT; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BIT);
procedure READ (L: inout LINE; VALUE: out BIT_VECTOR; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE :out BIT_VECTOR);
procedure READ (L: inout LINE; VALUE: out BOOLEAN; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out CHARACTER; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out CHARACTER);
procedure READ (L: inout LINE; VALUE: out INTEGER; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out INTEGER);
procedure READ (L: inout LINE; VALUE: out REAL; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out REAL);
procedure READ (L: inout LINE; VALUE: out STRING; GOOD out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out STRING);
procedure READ (L: inout LINE; VALUE: out TIME; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out TIME);
-- 标准类型的输出例行程序:
procedure WRITELINE (file F: TEXT; L: inout LINE);
procedure WRITELINE (L: inout LINE:VALUE: in BIT;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH :=0);
procedure WRITELINE (L: inout LINE:VALUE: in BIT_VECTOR;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH :=0);
procedure WRITELINE (L: inout LINE:VALUE: in BOOLEAN;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH :=0);
procedure WRITELINE (L: inout LINE:VALUE: in CHARACTER;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH :=0);
procedure WRITELINE (L: inout LINE:VALUE: in INTEGER;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH :=0);
procedure WRITELINE (L: inout LINE:VALUE: in REAL;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH :=0;
                DIGITS: in NATURAL:= 0);
procedure WRITELINE (L: inout LINE:VALUE: in STRING;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH :=0);
procedure WRITELINE (L: inout LINE:VALUE: in TIME;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH :=0;
                UNIT: in TIME: ns);
-- 文件位置预测:
-- function ENDFILE (file F: TEXT) return BOOLEAN;
end TEXTIO;
```

附录 D　专题设计项目

下面这些专题设计项目中的每一个都应该选择一个恰当的 FPGA 或 CPLD 作为目标器件，并按如下步骤对其加以实现：

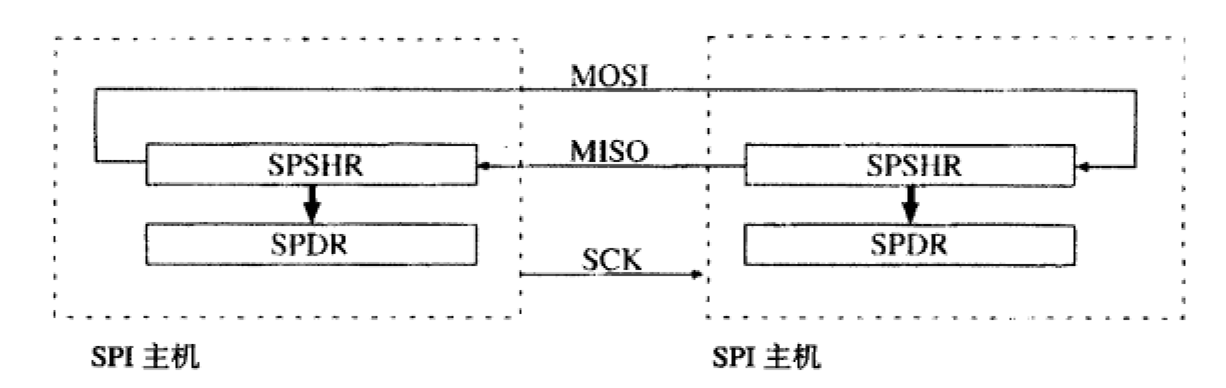
1. 做出系统的整体设计规划，并画出实现框图，如果需要的话，可以适当地把系统分为几个模块。针对每个模块，尽可能准确地提出一个算法，画出 SM 图或状态图。如果没有特殊指定，那么所设计的系统应该为具有恰当电路的同步系统，并且此系统的输入通过时钟进行同步。
2. 写出每个模块的可综合的 VHDL 代码，并对其进行仿真和编译。为了减少硬件的时序问题，用信号而不用变量，并且要保证所编写的代码在综合时不会生成锁存器。经过验证证明每个模块均可正常工作后，使用测试平台对编写的 VHDL 代码进行测试。
3. 对所编写的代码进行集成、仿真和测试。
4. 对所编写的代码进行适当的修改，并使用目标器件对其进行综合。综合后对系统进行仿真。
5. 为目标器件生成一个比特文件（bit file），并将其下载。验证硬件工作情况正确与否。

P1. 按键门锁

设计一个按键门锁，该门锁的输入采用标准电话键盘。在此模块中使用第 4 章中介绍的键盘扫描器。密码的长度为 4 到 7 个数字。当按下密码外加‘#’键后，门锁打开。只要一直按下‘#’键，门就保持打开状态。当‘#’键被释放时，门重新被锁上。如果要更改之前设定的密码，那么首先输入正确的前密码，然后按下‘*’键。这时，门锁进入到“存储”模式，“存储”指示灯点亮，直到密码修改完毕后指示灯才熄灭。然后我们键入新的密码并以‘#’键结尾，然后再输入一遍新的密码并以‘#’键结尾。如果第二遍输入的数字与第一遍输入的数字不符，那么需要再次输入两遍新的密码。密码可以存储在 8 个 4 位寄存器阵列中，或者存储在一个较小的 RAM 中。存储时，先存储‘#’键的代码，再存储 4 位密码的代码。同时，要求提供一个复位键，这个键不在键盘上。当按下复位键后，系统进入“存储”模式，此时可以输入一个新的密码。另外，用一个计数器计算进门时输入密码的次数。从键盘模块我们可以得到 3 个信号：一个 4 位密码、一个按键信号(*Kd*)和一个有效数据信号(*V*)。

P2. 同步串行外围接口

设计一个适用于微控制器的 SPI（同步串行外围接口）模块。该 SPI 用于同外围设备或其他微控制器进行串行同步通信。此 SPI 包含 4 个寄存器——*SPCR*（SPI 控制）、*SPSR*（SPI 状态）、*SPDR*（SPI 数据）和 *SPSHR*（SPI 移位寄存器）。下面的图展示了两个 SPI 是如何连接起来实现串行通信的。一个 SPI 作为 SPI 主机，另一个作为 SPI 从机。SPI 主机为同步传输和接收操作提供时钟。当一个字节的数据载入主 *SPSHR* 时，它开始串行传输并提供一个串行信号（*SCK*）。在 8 个时钟内，主移位寄存器中的数据与从移位寄存器中的数据对调。当传输完成时，*SPSHR* 中的每个数据都转移到相应的 *SPDR* 中。*SPSR* 中的 SPI 标志位（*SPIF*）置位。



管脚的功能取决于设备为主模式还是从模式：

MOSI——SPI 主机的输出，SPI 从机的输入
MISO——SPI 主机的输入，SPI 从机的输出
SCK——SPI 主机的输出，SPI 从机的输入

SPDR 和 *SPSHR* 均映射到同一地址。对此地址进行读操作可以从 *SPDR* 中读取数据，但是只能在 SPSHR 中写入数据。*SPSR* 的第 7 位为 SPI 标志位（*SPIF*）。*SPSR* 还包含错误标志位，但是在此设计中我们不考虑这些标志位。按如下顺序操作会使 *SPIF* 清零：

当 *SPIF* 置位时，读取 *SPSR*。
对 *SPDR* 地址进行读操作或写操作。

SPCR 寄存器包含的如下几位：

SPIE—— SPI 干扰使能
SPE——SPI 使能
MSTR——主模式时置'1'，从模式时置'0'

SPR1 和 SPR0——设置 SCLK 的速率，如下所示：
SPR1&SPR0 = 00　　*SCK* 的速率 = 系统时钟速率/2
SPR1&SPR0 = 01　　*SCK* 的速率 = 系统时钟速率/4
SPR1&SPR0 = 10　　*SCK* 的速率 = 系统时钟速率/16
SPR1&SPR0 = 11　　*SCK* 的速率 = 系统时钟速率/32

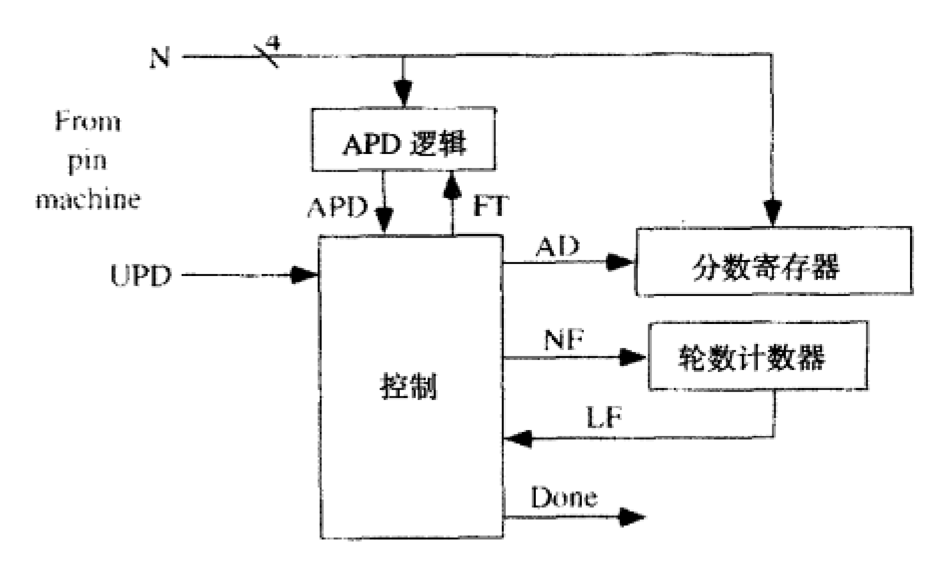
P3. 保龄球比赛记分器

下面设计的数字系统将用做保龄球比赛的记分器。此记分系统按照如下保龄球规则进行记分：保龄球比赛以局为单位，一局分 10 轮，每轮有两次投球机会。在每轮开始时，10 个瓶子均为竖立的。如果第一次投球就把 10 个木瓶全部击倒，则此轮记为“全中”。如果第一球只击倒部分木瓶，那么允许投第二次。如果第二次投球把其他的木瓶全部击倒，则此轮记为“补中”。否则，得分为此轮中击倒木瓶的总数。

比赛的总分等于击倒木瓶的总数加上“全中”和“补中”的奖励分数。一个“全中”所得的分数等于 10 分（此轮击倒了 10 个木瓶）再加下 2 次投球击倒的木瓶个数。一个“补中”所得的分数等于 10 分（此轮击倒了 10 个木瓶）再加下一次投球击倒的木瓶个数。如果在第 10 轮运动员掷出“补中”，那么他/她可以再投球一次。运动员所得总分等于这次投球击倒木瓶的个数加上当前所得分数。如果在第 10 轮运动员掷出“全中”，那么他/她可以再投球 2 次。运动员所得总分等于这 2 次投球击倒木瓶的个数加上当前所得分数。如果在第 9 轮和第 10 轮运动员都掷出“全

中"，那么他/她可以再投球两次，而且第一次奖励投球的分数在总分中加两次（一次为第 9 轮"全中"的奖励分数，一次为第 10 轮"全中"的奖励分数），第二次奖励投球所得分数只加一次。一局保龄球比赛的最高分为 300（均为"全中"）。下表为一次保龄球比赛的记分示例：

轮	第一次投掷	第二次投掷	结果	分数
1	3	4	7	7
2	5	5	补中	7 + 10 = 17
3	7	1	8	17 + 7（第 2 轮"补中"的奖励分数）+ 8 = 32
…	…	…	…	87
9	10	–	全中	87 + 10 = 97
10	10	–	全中	97 + 10（本次击倒 10 个木瓶）+ 10（第 9 轮"全中"的奖励分数）
–	6	3	–	117 + 6（第 9 轮"全中"的奖励分数） + 6（第 10 轮"全中"的奖励分数） + 3（第 10 轮"全中"的奖励分数）= 132



此记分系统框图如上图所示。控制网络有 3 个输入：*APD*（所有木瓶均被击倒）、*LF*（最后一轮）和 *UPD*（更新）。当所有的木瓶均被击倒时（不管是投球一次击倒全部木球，还是投球两次击倒全部木球），*APD* 为 1。当轮数计数器在状态 9 时（第 10 轮），*LF* 为 1。*UPD* 信号会使分数进行更新。在每次投掷后的一个时钟周期内 *UPD* 为 1。在两次更新之间存在很多个时钟周期。

控制网络有 4 个输出：*AD*，*NF*，*FT* 和 *Done*。*N* 表示当前投掷击倒的木瓶数。如果 *AD* 为 1，则 *N* 在下一个时钟上升沿到来时加到分数寄存器中。如果 *NF* 为 1，则在下一个时钟上升沿到来时轮数计数器加 1。在每轮投掷第一球时，*FT* 为 1。当所有 10 轮和奖励球全部投掷完毕后，*Done* 为 1。

使用一个 10 位寄存器存储分数，分数用 BCD 码（而不是二进制码）表示，因此分数 197 表示为 01 1001 0111。寄存器中分数的低两位数字（十进制数）使用两个 7 段 LED 指示器显示。当 *ADD* = 1 且寄存器时钟沿到来时，*N* 加到寄存器上。*N* 为 4 位二进制数，其取值范围为 0 到 10。我们使用一个 4 位 BCD 计数器模块保存中间的 BCD 数字。注意，在较低的 4 位，把一个二进制数加到一个 BCD 数字上，并给出一个新的 BCD 数字及进位。

P4. 简单的微型计算机

设计一个简单的 8 位有符号二进制数微型计算机。它使用键盘进行数据输入，并带有一个 256×8 的静态 RAM 存储器。此微型计算机含有以下几个 8 位寄存器：*A*（累加器）、*B*（乘法器）、*MDR*（内存数据计数器）、*PC*（程序指针）和 *MAR*（内存地址寄存器）。*IR*（指令寄存器）可以

是 5～8 位，其位数取决于指令编码的方式。寄存器 B 与寄存器 A 相连，因此在进行乘法运算时，这两个寄存器可以一起移位。只允许使用一个 8 位加法器和一个取补器。微型计算机包含一个 256 字×8 比特的内存，用来存储指令和数据。此微型计算机有两个模式：(a)内存载入，(b)程序执行。使用 DIP 开关进行模式选择。

内存载入模式工作方式如下：Select mode = 0 并重启系统。接着按下键盘上的两个键，然后再按下一个键，使每个字均载入内存，第一个字载入到地址 0，第二个字载入到地址 1，依次类推。数据在程序执行后立即载入内存。程序执行模式工作方式如下：Select mode = 1 并按下重启键。从地址 0 中的指令开始执行。

每个指令的长度为 1～2 个字。第一个字为操作码，第二个字（如果存在的话）为 8 位内存地址或立即操作数，我们可以通过操作码中的一个位辨别该字是内存地址还是立即操作数。负数用二进制补码表示。实现下列指令：

指令	说明
LDA<memadd>	从给定地址内存单元中取出来放入 A 寄存器。
LDA<imm>	把立即数放入 A 寄存器。
STA<memadd>	把 A 的内容存储在给定地址内存单元中。
ADD<memadd>	把给定地址内存单元中的数与 A 相加。如果存在进位，则把进位标志位置位。如果存在二进制补码向上溢出，则把 V 置位
ADD<imm>	把立即数与 A 相加。如果存在进位，则把进位标志位置位。如果存在向上溢出，则把 V 置位。
SUB<memadd>	从 A 中减去给定地址内存单元中的数。如果存在借位，则把借位标志位置位。如果存在二进制补码向下溢出，则把 V 置位。
SUB<imm>	从 A 中减去立即数。如果存在借位，则把借位标志位置位。如果存在二进制补码向下溢出，则把 V 置位。
MUL<memadd>	B 乘以给定地址内存单元中的数，结果放入 A & B。
MUL<imm>	B 乘以立即数。
SWAP	交换 A 和 B。
PAUSE	当按键按下和释放时间段内暂停（注意：A 寄存器总在 LED 上显示）。
JZ<target addr>	如果 A=0，则跳转到目标地址。
JC<target addr>	如果进位标志位(CF)置位，则跳转到目标地址。
JV<target addr>	如果溢出标志位(V)置位，则跳转到目标地址。

控制模块使用链接状态机实现，乘法器控制器用一个单独的状态机实现。状态数要尽可能少（较好的解决方案应含有大约 10 个状态）。乘法器控制器使用单独的计数器，以计算移位的个数。假设时钟速率足够慢，所以此微型计算机可以在一个时钟周期内完成存储器访问。

P5. 基于堆栈的计算器

设计一个用于 8 位有符号二进制数的基于堆栈的计算器。此计算器的输入数据可以通过键盘或具有单独按键的 DIP 开关键入。此计算器应该可以进行如下操作：

操作	说明
键入	把 8 位输入数据放入堆栈。
0-clear	把堆栈顶部清零，堆栈计数器复位，向上溢出复位，等等。

1-add　用堆栈顶部两个数据单元中数据的和取代原来的数据。

2-sub　用堆栈顶部两个数据单元中数据的差取代原来的数据（最顶部单元数据-下一个单元中的数据）。

3-mul　用堆栈顶部两个数据单元中数据的积取代原来的数据（8 位×8 位得到一个 8 位的积）。

4-div　用堆栈顶部两个数据单元中数据的商取代原来的数据(最顶部单元数据÷下一个单元中的数据)(8 位÷8 位得到一个 8 位的商)。

5-xchg　交换堆栈顶部两个数据单元的内容。

6-neg　用堆栈顶部两个数据单元中数据的补码取代原来的数据。

负数用二进制补码形式表示。二进制补码的向上溢出用向上溢出指示器显示。当积需要多于 8 位数表示（包括符号）或出现除以 0 的情况时，此指示器也应该置位。

实现一个大小为 4 个字（每个字 8 比特）的堆栈模块。此堆栈可以完成如下操作：对堆栈的头两个字进行入栈、出栈和交换操作。堆栈顶部第一个字中存储的内容总是使用 8 个 LED 显示出来。此堆栈中应该包含一个指示器以指出堆栈是否存在向上溢出（第 5 个字想要入栈）和向下溢出（对一个空的堆栈进行出栈操作或把堆栈顶部数据与一个内容为空的位置交换）。

此计算器设计控制单元使用链接状态机实现。画出主 SM 图，并且为乘法器控制器和除法器控制器单独画出 SM 图。当进行算术单元设计时，尽可能别加入不必要的寄存器。你可以使用三个寄存器（每个的大小为 8 或 9 比特）、一个加法器和两个取补器等实现此算术单元。

P6. 浮点数算术单元

设计一个浮点数算术单元。每个浮点数均由一个 4 位小数和一个 4 位指数表示，负数用二进制补码形式表示（这是第 7 章中使用的记数法）。要求此算术单元能够使用下面的浮点数指令：

001　FPL——向浮点数累加器中载入数据

010　FPA——在累加器中加上浮点操作数

011　FBS——从累加器中减去浮点操作数

100　FPM——用累加器乘以浮点操作数

101　FPD——（可选）浮点累加器除以浮点操作数

每个操作的结果（4 位小数和 4 位指数）均存储在浮点数累加器中。所有的输出都应该进行标准化。累加器中的数总是用十六进制数字表示，并通过 7 段 LED 展示出来。使用一个 LED 指示是否存在向上溢出。

浮点单元的输入为一个 4×4 的十六进制键盘，并使用与第 4 章中的扫描器相似的设备进行键盘读取。每条指令用 3 个十六进制数字表示——操作码、小数和指数。例如，可以把 FPA 1.011×2^{-3} 编码为 2 B D = 0010 1011 1101。假设所有的输入均为标准型数或零。设计应该包含以下模块：小数单元、分数单元、控制模块、4 位二进制码-7 段码显示转换逻辑。

P7. 井字游戏

使用 FPGA 设计一款井字游戏机。输入为一个 3×3 的键盘、一个复位按键和一个开关 SW1。若 SW1 断开，则只要游戏机可以赢，就判游戏机赢；否则判为和局（游戏双方均没有赢）。如果 SW1 闭合，则游戏机中有一部分逻辑就可以被通过，这样玩家有时会取得胜利。输出为 3×3 的

LED 阵列，每个方格中都带有一个红灯和一个绿灯。使用两个 LED 指出到底是玩家赢还是游戏机赢。如果为平局，那么所有的 LED 灯都亮。由于游戏机是守方，所以总是玩家先走。每当玩家走完后，游戏机等待 2 秒后再走。要求编写此同步数字系统的 VHDL 代码，并且要求实现的系统可以有效地利用硬件资源。

玩此游戏有这样一个策略（玩家 =X，游戏机 =O）：

1. 玩家先走。
2. 游戏机给出恰当的第一步。如果玩家从中心开始走，那么游戏机就从棋盘的角开始走，否则游戏机从中心开始走。
3. 在玩家走每一步后，游戏机应按照下面的顺序进行检查：
   (a)在同一行有两个 O：游戏机走同行内的第三个空格并赢得游戏。
   (b)在同一行有两个 X：游戏机走同行内的第三个空格，围堵玩家。
   (c)如果此步为游戏机的第二步，那么此步要进行特殊的规划：如果玩家的头两步在相对的两个角上，那么此步必须在边上。如果玩家第一步走在中间，那么只要不是(b)的情况，游戏机的第二步就走角。
   (d)两个交叉的行上各有一个 X：游戏机在两行的交叉方格落棋（为了阻止玩家胜利）。
   (e)如果没有更好的落棋位置，那么游戏机就随便走。

只有当想要落棋的方格是空着的时候，上面的规则才起作用。

P8. CORDIC 计算单元

CORDIC（坐标旋转数字计算机）是一种利用二维平面旋转计算三角方程的计算技术。此算法应用广泛，从计算器到全球定位系统中都有应用。由于此算法的计算只是反复的相加移位过程，所以对于数字系统来说此算法是很完美的。算法的说明详见论文① “A Survey of CORDIC Algorithms for FPGA-Based Computers”，http://www.andraka.com/files/crdcsrvy.pdf。

使用 FPGA 实现 CORDIC 算法。此设计必须能够准确得到输入角度为–179° ~+180° 时的 sin 和 cos 值。精度只要求为 8 位。通过键盘得到的输入为十进制格式的，为 3 个十进制数字和符号位（首先输入数字的最高有效位，符号在数据输入完毕后再输入）。最初，角度用 BCD 码表示，接着转化为二进制格式（负数用二进制补码表示）。给 sin 和 cos 指定两个按键。输出显示在 4 个 7 段数码管 LED 上。

下面的伪码展示了最基本的 CORDIC 算法。阅读上文给出的参考文献，并亲自研究算法的迭代过程，以便于更好地理解此算法。

```
for i = 0 to n                  //精度为n位
    dx= x / ( 2 ^ i )           //x为表示小数值的16位寄存器。其初始值为.607 (1001_1011_
                                //0111_0001)。算法完成后，x中为cos(a)。dx也为16位
    dy= y / ( 2 ^ i )           //y为表示小数值的16位寄存器。其初始值为0 (0000_0000_0000
                                //_0000)。算法完成后，y中为sin(a)。dy也为16位
    da= arctan ( 2 ^ -i )//查找表中提前计算的值应该按照如下方式表示：整个数的高8
                                //位，小数部分的低8位
```

① R.Andraka,"A Survey of CORDIC Algorithms for FPGA-Based Computers," in *Proceedings of the* 1998 *ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays* , pp.191-200,February 22-24,1998.

```
                              //a 为输入角度值，至少用 10 位表示
                              //所有的输入角度均为完整数（whole number）
    if (a >= 0) then
          x = x - dy;  a = a - da; y = y + dx;
    else
          x = x + dy;  a = a + da; y = y - dx;
    end if
end loop
```

使用此算法时，请注意算法不能给出与 sin 和 cos 相对应的负值和正值，所以需要另外设计逻辑电路确定符号。上面的算法只适用于输入角度为-90～+90 的情况。设计者可以把所有的计算都在第一象限进行，这样就可以简化设计[例如，sin(105) = sin(75)]。

P9．均值和标准差计算器

设计一个具有特殊用途的计算器，它可以计算一个考试分数集合的均值和标准差。输入从十进制键盘获得，输出为 LCD 显示器。每个考试分数均为整数，且取值范围为 0～100。数据的个数为 1～31 个。

输入序列：对于每个分数，先键入 1 个、2 个或 3 个数字，再键入 E（enter）。当所有的分数都键入后，按下 A 计算均值，再按下 D 计算标准差。均值和标准差均要求保留小数点后一位小数。

标准差公式为

$$\text{标准差} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - A)^2}{N}} = \sqrt{\frac{\sum_{i=1}^{N}{x_i}^2}{N} - A^2}$$

其中 A 为均值。我们使用后一个标准差公式，因为没有必要把 N 个分数都保存下来。

设计应该包含 3 个主要模块：输入、计算和显示。计算模块计算输入数据的均值和标准差。所有的计算都必须在二进制整数格式下完成。在输入模块输入数据乘以 10，并由十进制数转化为二进制数。在显示模块输出数据除以 10，并由二进制转化为十进制。

输入模块中应该包含键盘扫描器（与第 4 章中介绍的键盘扫描器相似）。每次当一个键被按下时，扫描器就为此键进行防颤抖和解码，然后输出一个与此键相对应的 4 位二进制数和一个有效信号（V）。输入模块会对键盘扫描器输出的数字进行处理并把输入数字转化为二进制数。输入模块应完成以下工作：

1. 如果输入数字为 0～9，则将其存入寄存器中。忽略无效输入。
2. 在键入 1 个、2 个或 3 个数字和 E 后，检查输入数据的取值范围是否满足要求（≤100）。如果不满足要求，则输出错误信号。
3. 如果输入数据满足取值范围，则在其后附加 BCD 数字 0，也就相当于把原数据乘以 10。例如，如果输入序列为 7, 9, E，则 BCD 寄存器中存储 0000 0111 1001 0000（790）。
4. 把 BCD 码转化为二进制码。当转换完成后，输出信号通知计算单元。
5. 当 A 键或 D 键被按下时，生成一个信号通知计算模块。

计算模块应包含两个寄存器，一个用来计算输入的累加和，另一个用来计算输入的平方的累加和。数据输入应为二进制整数，其十进制取值范围为 0～1000（分数×10）。假设 3 个输入控制

信号为 V1（有效数据）、A（计算和输出均值）和 S（计算和输出标准差）。当均值计算未完成时，忽略 S。计算模块应该包含一个平方根电路，此电路可以计算一个 18 位二进制整数的平方根，并给出一个 9 位整数解。二进制平方根算法详见参考文献[35]。对计算模块进行测试时，要保证包含最坏的情况：31 个输入的最大均值（标准差为 0），30 个输入的最大标准差（均值为 500）。

显示模块驱动一个 2 线 LCD 显示器。此模块完成两个功能：首先它可以在输入数据的同时显示出输入的数字；第二它可以显示均值和标准差。在输入过程中，每个有效的十进制数字都会被移入显示器。当按下 E 时，输入数据仍显示在显示器上直到其他键被按下。在均值计算完毕后，显示模块将其转换为 BCD 码并通过第一条线输出到 LCD 显示器上。在标准差计算完毕后，显示模块将其转换为 BCD 码并通过第二条线输出到 LCD 显示器上。

P10. 四功能十进制计算器

设计一个具有 4 个功能的十进制手掌计算器，并用 FPGA 加以实现。输入为一个键盘，输出为 LCD 显示器。由于手掌计算器对速度要求不高，所以当用 FPGA 实现此设计时，应对其所占面积进行优化。此计算器的基本操作应与标准四功能计算器相似。

计算器主输入键盘有 16 个按键，标示如下：

```
7  8  9  ÷
4  5  6  *
1  2  3  –
0  .  =  +
```

另外还需要一个提供清零功能的按键。此计算器最多可以输入和输出 8 个十进制数字、一个小数点和一个符号（可选）。假设在任何时刻任何键都可能被按下。在键被按下时，计算器可以进行正确的操作，也可以选择忽略。如果输入多于 8 个数字，多出的数字就会被忽略。

如果答案多于 8 个数字，则对小数点最右边的数字进行截尾操作。

示例：123.45678 + 12345.678 = 12469.134

如果小数点左边的数字多于 8 个，则显示字母 E，表示有错误。对于小于 1 的数，在小数点前显示 0。

此计算器应该有三个模块。输入模块进行键盘扫描、防颤抖和译码。主模块接收数字和来自输入模块的命令并对它们进行处理。显示模块在 LCD 显示器上显示输入数据和计算结果。

主模块中有两个 8 数字 BCD 寄存器 A 和 B。寄存器 A 中有两个计数器和一个符号触发器（signA），其中一个计数器用来计算输入数字的个数（ctrA），另一个计数器用来计算十进制小数点右边数字的个数（rctA）。寄存器 B 中的硬件同寄存器 A 相似。当输入一个十进制数字时，此数字的 BCD 码就会被移入寄存器 A。每个计算的结果也存储在寄存器 A 中。显示模块总是显示寄存器 A 中的内容以及小数点和符号。当一个新数据的一个数字移入寄存器 A 时，之前寄存器 A 中的内容就转移到寄存器 B 中。虽然输入和输出均为有符号 BCD 码，但是在内部计算时均采用二进制补码算术。A 加 B 的典型计算过程如下：

1. 把 A 和 B 的小数点对齐。
2. 把 A 和 B 转化为二进制数（Abin 和 Bbin）。
3. 求 Abin 与 Bbin 的和。

4. 把结果转化为 BCD 码，并存储在 A 中。

5. 如果有向上溢出，则尽可能的对其进行纠正。如果不能纠正，则将 E（错误）标志位置位。

显示模块输出信号到 LCD 并正确地显示寄存器 A 中的内容。在 LCD 初始化和清零后，如果错误触发器置位，则显示 “E”。否则，如果 signA='1'，则输出负号和 8 位十进制数字，小数点应该在正确的位置。最高位的 0 用空位代替。



索　　引

B

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z



参 考 文 献

参考文献 14，19，21，27，28，39，41，46 和 48 为数字逻辑和数字系统设计的基本材料。参考文献 2，3，4，15，16，20，24，30，40，42，44，47，49 和 50 给出了关于 PLD，FPGA 和 CPLD 的相关信息。参考文献 10，21，31，38，43，45 和 52 为 VHDL 语言的基本说明。参考文献 5，6，8，9，17，18，22，23，33，34 和 51 为 VHDL 语言的高级说明。参考文献 1，7，11，29，32 和 36 介绍了硬件测试和可测试设计的相关知识。参考文献 13，25，26 和 37 介绍了 MIPS ISA 和 MIPS 处理器结构。参考文献 37 对各种计算机结构进行了很详细的介绍。如果理解了这篇参考文献，那么对于学习本书中的第 9 章是很有帮助的。

1. Abromovici, M., Breuer, M., and Friedman, F. *Digital Systems Testing and Testable Design*. Indianapolis, Ind. Wiley–IEEE Press, 1994.
2. Actel Corporation, Actel Technical Documentation, www.actel.com/techdocs/
3. Altera Corporation, Altera Literature, www.altera.com/literature/lit-index.html
4. Atmel Corporation, Atmel Products, www.atmel.com/products
5. Armstrong, James, and Gary, G. *Structured Logic Design with VHDL*. Upper Saddle River, N.J.: Prentice Hall, 1993.
6. Ashenden, Peter J. *The Designer's Guide to VHDL*, 2nd ed. San Francisco, Calif.: Morgan Kaufmann, an imprint of Elsevier, 2002.
7. Bardell, P. H., and McAnney, W. H. "Self-Testing of Logic Modules," *Proceedings of the International Test Conference*, Philadelphia, PA November, 1982, pp. 200–204.
8. Berge, F., and Maginot, R. J. *VHDL Designer's Reference*. Boston, Mass.: Kluwer Academic Publishers, 1992.
9. Bhasker, J. *A Guide to VHDL Syntax*. Upper Saddle River, N.J.: Prentice Hall, 1995.
10. Bhasker, J. *A VHDL Primer*, 3rd ed. Upper Saddle River, N.J.: Prentice Hall, 1999.
11. Bleeker, H., van den Eijnden, P., and de Jong, Frans. *Boundary Scan Test—A Practical Approach*. Boston, Mass.: Kluwer Academic Publishers, 1993.
12. Brayton, Robert K. et al. *Logic Minimization Algorithms for VLSI Synthesis*. Boston, Mass.: Kluwer Academic Publishers, 1984.
13. Britton, Robert. *MIPS Assembly Language Programming*. Upper Saddle River, N.J.: Prentice Hall, 2003.
14. Brown, Stephen and Vranesic, Zvonko. *Fundamentals of Digital Logic with VHDL Design*, 2nd ed. New York: McGraw-Hill, 2005.
15. Brown, Stephen D., Francis, Robert J., Rose, Jonathan, and Vranesic, Zvonko G. *Field-Programmable Gate Arrays*. Boston, Mass.: Kluwer Academic Publishers, 1992.
16. Chan, P., and Mourad, S. *Digital Design Using Field Programmable Gate Arrays*. Upper Saddle River, N.J.: Prentice Hall, 1994.
17. Chang, K. C. *Digital Design and Modeling with VHDL and Synthesis*. Los Alamitos, Calif.: IEEE Computer Society Press, 1997.
18. Cohen, Ben. *VHDL—Coding Styles and Methodologies*. Boston, Mass.: Kluwer Academic Publishers, 1995.
19. Comer, David J. *Digital Logic and State Machine Design*, 3rd ed. New York: Oxford University Press, 1995.
20. Cypress Semiconductor Programmable Logic Documentation, www.cypress.com
21. Dewey, Allen. *Analysis and Design of Digital Systems with VHDL*. Toronto, Ontario, Canada: Thomson Engineering, 1997.

22. *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)*. New York: The Institute of Electrical and Electronics Engineers, 1993.
23. *IEEE Standard VHDL Language Reference Manual*. New York: The Institute of Electrical and Electronics Engineers, 1993.
24. Jenkins, Jesse H. *Designing with FPGAs and CPLDs*. Upper Saddle River, N.J.: Prentice Hall, 1994.
25. Kane, Gerry, and Heinrich, Joseph. *MIPS RISC Architecture*. Upper Saddle River, N.J.: Prentice Hall, 1991.
26. Kane, Gerry. *MIPS RISC Architecture*. Upper Saddle River, N.J.: Prentice Hall, 1989.
27. Katz, Randy H. *Contemporary Logic Design*, 2nd ed. Upper Saddle River, N.J.: Prentice Hall, 2004.
28. Kohavi, Z., *Switching and Finite Automata Theory*. New York: McGraw-Hill, 1979.
29. Larsson, Erik *Introduction to Advanced System-on-Chip Test Design and Optimization*. Springer, 2005.
30. Lattice Semiconductors, www.latticesemi.com
31. Mazor, Stanley, and Langstraat, Patricia. *A Guide to VHDL*, 2nd ed. Boston, Mass.: Kluwer Academic Publishers, 1993.
32. McCluskey, E. J. *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Upper Saddle River, N.J.: Prentice Hall, 1986.
33. Navabi, Zainalabedin. *VHDL—Analysis and Modeling of Digital Systems*, 2nd ed. New York: McGraw-Hill, 1997.
34. Ott, Douglas E., and Wilderotter, Thomas J. *A Designer's Guide to VHDL Synthesis*. Boston, Mass.: Kluwer Academic Publishers, 1994.
35. Parhami, Behrooz. *Computer Arithmetic: Algorithms and Hardware Design*. New York: Oxford University Press, 2000.
36. Parker, Kenneth P. *The Boundary Scan Handbook*, 3rd ed. New York: Springer, 2003.
37. Patterson, David A., and Hennessey, John L. *Computer Organization and Design: The Hardware Software Interface*, 3rd ed. San Francisco, Calif.: Morgan Kaufmann, an imprint of Elsevier, 2005.
38. Perry, Douglas. *VHDL: Programming by Example*, 4th ed. New York: McGraw-Hill, 2002.
39. Prosser, Franklin P., and Winkel, David E. *The Art of Digital Design: An Introduction to Top-Down Design*, 2nd ed. Englewood Cliffs, N.J.: Prentice Hall, 1987.
40. QuickLogic Corporation, Products and Services, www.quicklogic.com
41. Roth, Charles H. *Fundamentals of Logic Design*, 5th ed. Toronto, Ontario, Canada: Thomson Engineering, 2004. (Includes Direct VHDL simulation software.)
42. Rucinski, Andrzej, and Hludik, Frank. *Introduction to FPGA-Based Microsystem Design*. Texas Instruments, 1993.
43. Rushton, Andrew, *VHDL for Logic Synthesis*, 2nd ed. New York: John Wiley & Sons Ltd., 1998.
44. Salcic, C., and Smailagic, A. *Digital Systems Design and Prototyping Using Field Programmable Logic*, 2nd ed. Boston, Mass.: Kluwer Academic Publishers, 2000. (Includes VHDL software for Altera products.)
45. Skahill, Kenneth, and Cypress Semiconductor. *VHDL for Programmable Logic*. Reading, Mass.: Addison-Wesley, 1996. (Includes VHDL software for Cypress products.)
46. Smith, M. J. S. *Application-Specific Integrated Circuits*. Reading, Mass.: Addison Wesley, 1997.
47. Tallyn, Kent. "Reprogrammable Missile: How an FPGA Adds Flexibility to the Navy's TomaHawk," *Military and Aerospace Electronics*, April 1990. Article reprinted in *The Programmable Gate Array Data Book*, San Jose, Cal.: Xilinx, 1992.

48. Wakerly, John F. *Digital Design Principles and Practices*, 4th ed. Upper Saddle River, N.J.: Prentice Hall, 2006.
49. Xilinx, Inc. *Xilinx Documentation and Literature*, www.xilinx.com/support/library.htm
50. XILINX, Inc. *The Programmable Logic Data Book*, 1996. www.xilinx.com
51. Yalamanchili, S. *Introductory VHDL: From Simulation to Synthesis*. Upper Saddle River, N.J.: Prentice Hall, 2001. (Includes XILINX Student Edition Foundation Series Software.)
52. Yalamanchili, S. *VHDL: A Starter's Guide*, 2nd ed. Upper Saddle River, N.J.: Prentice Hall, 2005.

